# ABSTRACT

Title of dissertation: PLANETARY SCALE DATA STORAGE

Benjamin Bengfort
Doctor of Philosophy, 2018

Dissertation directed by: Professor Peter J. Keleher
Department of Computer Science

The success of virtualization and container-based application deployment has fundamentally changed computing infrastructure from dedicated hardware provisioning to on-demand, shared clouds of computational resources. One of the most interesting effects of this shift is the opportunity to localize applications in multiple geographies and support mobile users around the globe. With relatively few steps, an application and its data systems can be deployed and scaled across continents and oceans, leveraging the existing data centers of much larger cloud providers.

The novelty and ease of a global computing context means that we are closer to the advent of an Oceanstore, an Internet-like revolution in personalized, persistent data that securely travels with its users. At a global scale, however, data systems suffer from physical limitations that significantly impact its consistency and performance. Even with modern telecommunications technology, the latency in communication from Brazil to Japan results in noticeable synchronization delays that violate user expectations. Moreover, the required scale of such systems means that failure is routine.

To address these issues, we explore consistency in the implementation of distributed logs, key/value databases and file systems that are replicated across wide areas. At the core of our system is hierarchical consensus, a geographically-distributed consensus algorithm that provides strong consistency, fault tolerance, durability, and adaptability to varying user access patterns. Using hierarchical consensus as a backbone, we further extend our system from data centers to edge regions using federated consistency, an adaptive consistency model that gives satellite replicas high availability at a stronger global consistency than existing weak consistency models.

In a deployment of 135 replicas in 15 geographic regions across 5 continents, we show that our implementation provides high throughput, strong consistency, and resiliency in the face of failure. From our experimental validation, we conclude that planetary-scale data storage systems can be implemented algorithmically without sacrificing consistency or performance.

# PLANETARY SCALE DATA STORAGE

by

Benjamin Bengfort

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Peter J. Keleher, Chair/Advisor
Professor Dave Levin
Professor Amol Deshpande
Professor Daniel Abadi
Professor Derek C. Richardson

# Preface

If needed.

# Foreword

If needed.

# Dedication

To Irena and Henry.

# Acknowledgments

I could not have done this alone.

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| CC | Causal Consistency |
| COW | Copy on Write |
| EC | Eventual Consistency |
| HC | Hierarchical Consensus |
| LIN | Linearizablity |
| LWW | Last Writer Wins |
| SC | Sequential Consistency |
| WC | Weak Consistency |

# Chapter 1: Introduction

Eighteen years ago the Oceanstore paper [1] presented a vision for a data utility infrastructure that spanned the globe. The economic model was that of a cooperative utility provided by a confederation of companies that could buy and sell capacity to directly support their users, with regional providers like airports and cafes installing servers to enhance performance for a small dividend of the utility. This economic model meant that Oceanstore's requirements centered around an untrusted infrastructure to support nomadic data: connectivity, security, durability, and location agnostic storage. To meet these requirements, the Oceanstore architecture was composed of two tiers: pools of byzantine quorums that made localized consistency and placement decisions, along with an optimistic dissemination tree layer that moved data between quorums as correctly as possible without providing guarantees. This architecture, along with a reliance on encryption and key-based access control, could facilitate grid computing storage, a truly decentralized and independent participation of heterogenous computational resources across the globe [2].

Unforeseen by Oceanstore, however, was a fundamental shift in how companies and users accessed computing infrastructure. Improvements in virtualization

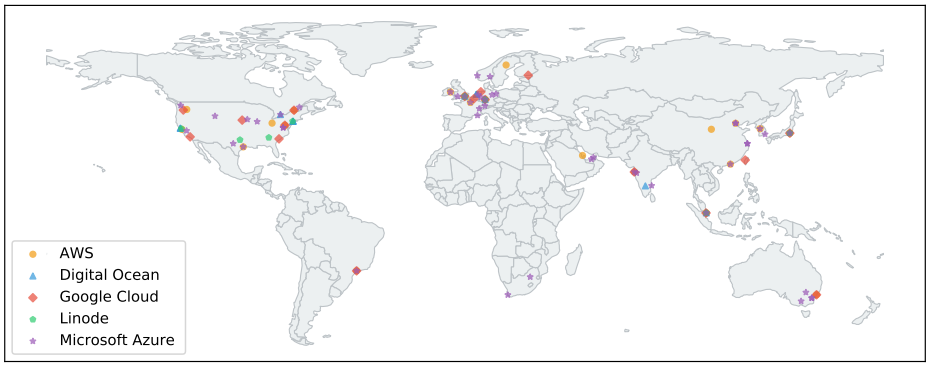


Figure 1.1: Data center locations of popular cloud providers span the globe, creating the possibility for the deployment of geo-replicated data storage inside of a trusted infrastructure without Byzantine failure.

management [3] and later container computing [4] allowed big Internet companies to easily lease their unused computational resources and disk capacity to application developers, making cloud computing [5] rather than independent hardware purchasing and hosting the norm. Furthermore, from smarter phones to tablets and netbooks, user devices have become increasingly mobile with reduced disk capacities, which meant that applications, from photos to email and contacts, primarily store user data in cloud silos. The cloud economy means that there is a *trusted infrastructure* of virtual resources that span globe, provisioned by a single provider as shown in Figure 1.1.

So what does cloud computing mean for the requirements and assumptions of Oceanstore? First, Oceanstore's strict requirements for security that meant per-user encryption and byzantine agreement between untrusted servers can be relaxed to application and transport-level encryption and non-byzantine consensus supported by authenticated communication. Second, the requirements for performance have

dramatically increased as ubiquitous computing has become the norm and as more non-human users are participating in networks. Increased capacity, however, cannot come at the cost of correctness or consistency, and the increased rate of requests means that asynchronous commits and conflict resolution become far more difficult. For these reasons, we believe that a vision for an Oceanstore today would focus on *consistency* rather than security.

Consistent behavior in distributed systems becomes increasingly complex to implement and reason about as the system size grows and requires increased coordination. In 2021, Cisco forecasts over 25 billion devices will contribute to 105,800 GBps of global Internet traffic, 26% of which will be file sharing and application data, and 51% of which will originate from machine to machine-to-machine applications [6]. New types of networks including sensor networks [7], smart grid solutions [8], self-driving vehicle networks [9], and an Internet of Things [10] will mean an update model with many publishers, few subscribers, and increasingly distributed accesses. To support this growth and facilitate speed, traffic is consistently moving closer to the edge; Cisco predicts that cross-continent delivery will drop from 58% of traffic in 2016 to 41% in 2021 and that metro delivery will grow from 22% to 35% [6]. Localization means that the cloud will be surrounded by a fog of devices that participate in systems by contributing data storage and computation to an extent greater than access-oriented clients might.

Based on these trends and inspired by the work of Oceanstore, we propose that a planetary-scale data storage system would be made up of an adaptable, two tier architecture of both cloud and fog infrastructure. The first tier, in the cloud, would

be a strong consistency, fault tolerant and highly resilient geo-replicated consensus backbone. The second tier, via the fog, would be a high availability heterogenous network with a relaxed consistency model that could quickly disseminate updates. Such a system would be difficult to manually manage, therefore both tiers and the system as a whole would also have to automatically monitor and adapt to changes in access patterns and node and network availability during runtime. Our proposed design is therefore a flexible data fabric with three facets: geo-replicated consensus, high-availability dissemination and accesses, and online adaptability and optimization.

In this dissertation we explore all three of these facets that comprise a robust geo-distributed data system. Because consensus algorithms have not been designed to scale to large quorum sizes we propose *hierarchical consensus*, as the first tier consensus backbone that can scale to hundreds or thousands of nodes while maintaining strong consistency guarantees. To implement the second tier, we propose *federated consistency*, a hybrid consistency model that allows heterogenous replicas varying guarantees depending on application requirements by integrating strong and eventual consistency models. Finally, we propose *adaptive consistency*, a model for emergent introspection that uses localized machine learning to optimize single-replica behavior which, when taken together, leads to an increase in the overall performance of the system. We hypothesize that the combination of hierarchical consensus, federated consistency, and adaptive monitoring lay out a foundation for truly large scale data storage systems that span the planet. The contributions of this dissertation are therefore as follows:

1. We present the design, implementation, and evaluation of hierarchical consensus, a consensus protocol that can scale to dozens or hundreds of replicas across the wide area.

2. We also investigate the design and implementation of federated consistency, a hybrid consistency model that allows strong, consensus-based systems to integrate with eventually-consistent, highly available replicas and evaluate it in a simulated heterogenous network.

3. We show the possibilities for machine learning-based system adaptation with a reinforcement learning approach to anti-entropy synchronizations based on accesses.

4. We validate our system by describing the implementation of a planetary-scale key-value data store and file system using both hierarchical consensus and federated consistency.

The rest of this dissertation is organized as follows. In the next chapter we will more thoroughly describe our proposed geo-replicated architecture by describing the motivations and challenges for our work and exploring case-studies of existing systems. Next, we will focus on the core backbone of our system, hierarchical consensus, and describe a globally fault tolerant approach to managing accesses to objects in the wide area. Using hierarchical consensus as a building block, the next chapter will focus on federated consistency, specifying a hybrid, heterogenous consistency model in the fog that interacts with the cloud consensus tier. At this

point we will have enough background to introduce our system implementation and describe our file system and key-value store. From there, we will explore learning systems that monitor and adapt the performance of the system at runtime, before concluding with related work and a discussion of our future research.

# Chapter 2: A Geo-Distributed System Architecture

The advent of cloud computing has accelerated both commercial and academic interest in distributed systems connected via wide area networks and the Internet. Cloud computing exists because large Internet companies, which had deployed extremely large data centers around the world to meet global user demand for their services, created extra capacity that could be leased to tenants on-demand. The global nature of cloud providers means that there is an opportunity for more common usage of geographically replicated data systems besides the specialized systems they developed for internal use. Though these systems have been made available to customers as provisioned services, they suffer from application-specific data models to narrow to solve general problems.

The specialization of the current generation of distributed systems is designed to optimize their behavior when deployed within a pristine data center context. This environment, with strong facility support and backbone communications, allows design choices that optimistically assume that repairs will be made quickly and that redundancy need only protect from few failures at a time [11–14]. This has led to a general architecture for geo-replication that provisions consistency requirements across transactions, individual objects, and within blocks stored on disk,

often leading to multiple, independent consistency models within the same system. This design leads to confusion about where data is stored and what guarantees can be made about each access.

Instead, a single, global consistency model is required to correctly reason about a single, global system's behavior. The central thesis of this dissertation is that this can only be achieved with a globally-distributed consensus protocol.

Consistency depends on the network environment and in highly curated data centers, systems are built using localized consensus [15]. Outside the data center, however, single process failures show the impossibility of distributed consensus [16]. Therefore our approach is a continuous, flexible consistency model with geographic consensus at its core and a federated approach at its edge. The result is a single, understandable consistency model that leads to an architecture capable of supporting global systems both inside and outside of the data center.

In this chapter, we will describe the details of our proposed architecture for a planet-wide distributed data store. To motivate our architectural decisions, we first describe the motivations and challenges for the design of such a system. We motivate our work in two parts. First, we argue that there is a new software deployment paradigm that requires geographic replication. Second, we argue that existing geo-replicated data stores are not general enough to meet the needs of that paradigm. We follow these motivations with the challenges of geo-replication described as requirements, which we will then use to present an overview of our system architecture.

## 2.1 Challenges and Motivations

Many of the world's most influential companies grew from the ashes of the dot-com bubble of the 1990s, which paid for an infrastructure of fiber-optic cables, giant server farms, and research into mobile wireless networks [17]. As these companies filled market voids in eCommerce, search, and social networking, they created new database technologies to leverage the potential of underused computational resources and low latency/high bandwidth networks that connected them, eschewing more mature systems that were developed with resource scarcity in mind [18, 19]. What followed was the rise and fall of NoSQL data systems, a microcosm of the proceeding era of database research and development [20].

Although there are many facets to the story of NoSQL, what concerns us most is the use of NoSQL to create geographically distributed systems, as these systems paved the way to the large-scale storage systems in use today. The commercial and open source interest in geo-replicated systems both for big data analysis and Web application development has led to the development of many database and file systems. These systems share common traits, allowing us to describe a general architecture for distributed systems. More importantly, the prevalence of such systems has led to a new application development paradigm: modern software must be designed as a global services.

### 2.1.1 A New Application Development Paradigm

The launch of the augmented reality game Pokémon GO in the United States was an unmitigated disaster [21]. Due to extremely overloaded servers from the release's extreme popularity, users could not download the game, login, create avatars, or find augmented reality artifacts in their locales. The company behind the platform, Niantic, scrambled quickly, diverting engineering resources away from their feature roadmap toward improving infrastructure reliability. The game world was hosted by a suite of Google Cloud services, primarily backed by the Cloud Datastore [22], a geographically distributed NoSQL database. Scaling the application to millions of users therefore involved provisioning extra capacity to the database by increasing the number of shards as well as improving load balancing and autoscaling of application logic run in Kubernetes [23] containers.

Niantic's quick recovery is often hailed as a success story for cloud services and has provided a model for elastic, on demand expansion of computational resources. A deeper examination, however, shows that Google's global high speed network was at the heart of ensuring that service stayed stable as it expanded [24], and that the same network made it possible for the game to immediately become available to audiences around the world. The original launch of the game was in 5 countries – Australia, New Zealand, the United States, the United Kingdom, and Germany. The success of the game meant worldwide demand, and it was subsequently expanded to over 200 countries starting with Japan [25]. Unlike previous games that were restricted with region locks [26], Pokémon GO was a truly interna-

tional phenomenon and Niantic was determined to allow international interactions in the game's feature set, interaction which relies on Google's unified international architecture and globally distributed databases.

Stories such Niantic's deployment are increasingly becoming common and medium to large applications now *require developers to quickly reason about how data is distributed in the wide area, different political regions, and replicated for use around the world.*

It is not difficult to find many examples of companies and applications, from large to small, that have international audiences and global deployments which highlight the new challenges of software development. Dropbox has users in over 180 countries and is supported in 20 languages, maintaining offices in 12 locations from Herzliya to Sydney [27]. Slack serves 9 million weekly active users around the world and has 8 offices around the world, prioritizing North America, Europe, and Pacific regions [28]. WeWork provides co-working space in 250+ international applications and uses an app to manage global access and membership [29]. Tile has sold 15 million of its RFID trackers worldwide and locates 3 million unique items a day in 230 countries [30]. Trello, a project management tool, has been translated into 20 languages and has 250 million world-wide users in every country except Tuvalu; their international rollout focused on marketing and localization [31]. Runkeeper [32] and DarkSky [33] are iOS and Android apps that have millions of global users and struggled to make their services available in other countries, but benefitted from international app stores. Signal and Telegraph, both encrypted messaging apps, have grown primarily in countries at the top of Transparency International's Corruption

Perception Index [34].

The new application development paradigm, even for small applications, is to build with the thought that your application will soon be scaling across the globe. None of the applications described above necessarily have geography-based requirements in the same way that an augmented reality or airline reservations application might, just a large number of users who regularly use the app from a variety of geographic locations. Web developers are increasingly discussing and using container based approaches both for development and small-scale production. Web frameworks have built in localization tools that are employed by default. Services are deployed on autoscaling cloud platforms from the start.

To address this paradigm shift, cloud service providers have expanded their offerings to include use of their distributed data stores to application developers, as in the case of Niantic. This presents two problems, however. First, those systems were designed for the huge applications of the Internet companies themselves, not for the general needs of a large audience of developers. Second, cloud providers organized their services around geographically distinct regions, allowing their tenants the choice to deploy their applications in one or more regions. As a result, tenants naturally choose cloud regions based on the locations of their users and treat regions as independent deployments from both a software and billing perspective. Even though there may be some cross-region backup to prevent catastrophic failures, region-specificity usually means that services are deployed piecemeal and partitioned.

Region-based organization of cloud services has ensured that users are able

to minimize latencies and provide applications to areas of interest, but applications have increasingly become more global. The possibility of region-agnostic deployment is tantalizing, particularly as larger applications spend a non-trivial amount of administration time determining where writes to objects are going to optimize their systems. Consistent updates across regions are not even considered as a possibility because in-house cloud services used data models that avoided large latencies wherever possible. Moreover, as we will see in the next section, the design of provisioned cloud databases have made it difficult to reason about consistent behavior. Not only is there a need for strong consistency semantics, data-location awareness, and geo-replication in distributed data storage systems, there is also the need for a familiar and standardized storage API.

### 2.1.2 Building Geo-Replicated Services

The growth of database systems distributed across the wide-area started with large Internet companies like Yahoo [35], Google [36], and Amazon [37] but quickly led to academic investigations. One reason that the commercial systems enjoyed this academic attention was that at the time, the unique scale of their usage validated the motivation behind their architecture. However their success has meant that these types of scales are no longer limited to huge software systems. The development of geo-replicated services has therefore undergone several phase shifts, and has led to a general framework that underlies most current systems.

The first shift was the creation of highly available, sharded systems intended

to meet the demand of increasing numbers of clients. Commercially, these types of systems include Dynamo [37] and BigTable [36], which in turn spawned open source and academic derivatives such as Cassandra [38] and HBase [39]. Although these systems did support many concurrent accesses, they achieved their availability by relaxing consistency, which many applications found to be intolerable. The second shift was a return to stronger consistency, even at the cost of decreased performance or expensive engineering solutions. Again, commercial systems led the way with Megastore [40] and Spanner [41] along with academic solutions such as MDCC [42] and Calvin [43, 44]. Part of this realignment was a reconsideration of the base assumptions that drove the NoSQL movement as expressed in the CAP theorem [45, 46]. The new thinking is that the lines between availability, partition tolerance, and consistency may not be as strictly drawn as previously theorized [47–49]. This has led to a final shift, the return of SQL, as the lessons learned during the first two phases are applied to more traditional systems. As before, both commercial systems, such as Aurora [50] and Azure SQL [51], and open source systems such as Vitess [52] and CockroachDB [53] are playing an important role in framing the conversation about consistency in this phase.

These systems have defined several strategies from relaxed consistency to interval time that are essential to understanding geo-replicated services. The first and primary strategy, however, is to shard data into independent groups of semi-related objects. A shard can be specifically defined either as buckets of objects, tablets of contiguous rows, or in the most extreme cases, individual objects.

Sharding allow extremely massive databases and file systems to be broken

down into smaller, related pieces that are more easily managed in a distributed context. Sharding allows concurrent access to objects without synchronization. If a shard is defined within a specific region, then it is easy to prioritize local accesses and coordination with that region. Sharding also provides a data model for underlying redundant storage. If a tablet can be written to a specific page on disk, that page can also be replicated to more easily colocate replicas with data. If multiple shards need to be accessed simultaneously in a transaction, then only the shards involved in the transaction need be coordinated, while all other shards can remain independent.

The unit of coordination is therefore at the shard level. The namespace of the system must be allocated to individual components of the system, which must be globally available, and requires coordination to move part of the namespace from one region to another. Accesses to data on disk for individual objects must also be allowed to happen in a fault-tolerant manner, which requires coordination between several replicas to guarantee no data loss. Finally, transactions that require access to multiple objects must be coordinated to ensure atomic guarantees. To support all of these features in a system, a multi-process architecture of independent components for geo-replication is generally deployed, as shown in Figure 2.1.

Clients access geographically-replicated data systems either by making geographic based requests via domain name (e.g. requesting a `.ca` addressed service vs. a `.fr` address) or by using IP and ping based network location [54]. Multiple, concurrent requests are load-balanced to container based compute nodes that hold application logic, elastically scaled to meet changing demand [55]. Though this aspect of distributed applications is outside the scope of a geo-replicated data store,

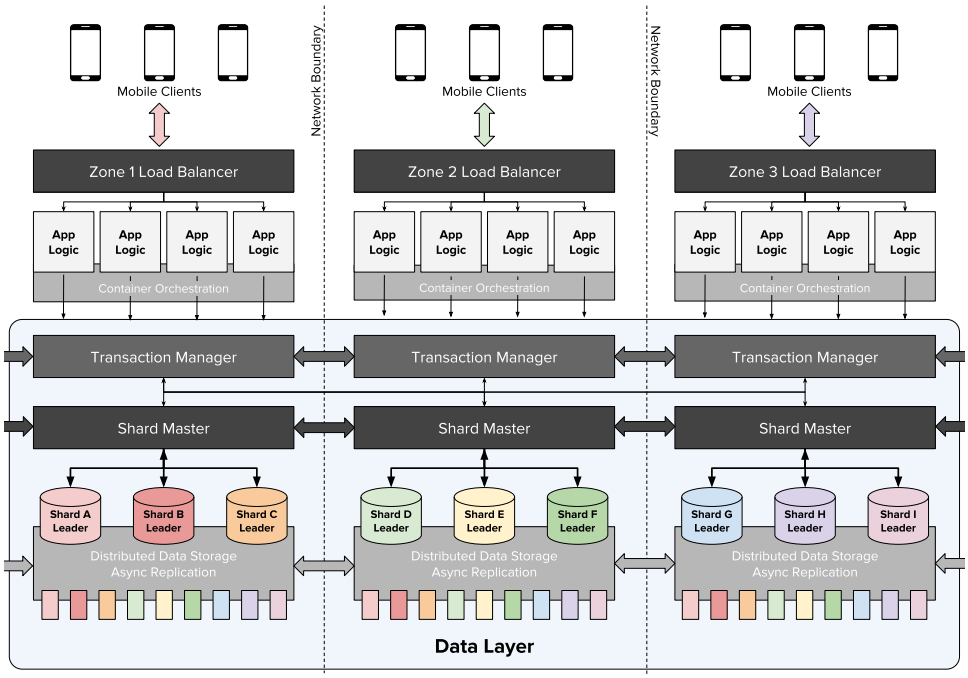


Figure 2.1: A generalized architecture of a geographically distributed application using a basic sharding strategy. The namespace of the database is coordinated by shard masters, which point to quorums of replicas who replicate data over multiple disks. Accesses to multiple objects are coordinated via transaction managers.

the increased frequency of accesses across multiple regions increases the likelihood of collisions – concurrent access by multiple clients to the same object(s), which leads to the consistency concerns of this dissertation.

The data layer follows the application logic layer and is the layer that must coordinate accesses from many simultaneous geographies. If the data layer supports transactions [43,44,47,50,52,53] then the top level of coordination is the transaction manager, which is responsible for identifying the shards that manage the objects and correctly committing or rolling back the transaction. Other systems support snapshot isolation for read-transactions [40,41], ensuring that all reads for a specified time window are consistent.

If the data layer does not support transactions or if only a single object is being accessed, then the system must coordinate with the shard master, a process that allocates the namespace to the replicas that manage those objects. Some systems use the data-model directly, using key-space addressing to determine the locality of objects [36, 41], others use consistent hashing [56, 57] to balance objects around a hash ring, coordinating the insertion and removal of name management servers. However, if the preservation of data locale and the ability to move objects between regions is required, then a synchronized lookup table must be used. The most popular mechanism to achieve namespace synchronization is to use a lock service such as Chubby [58] or etcd [59] to hand out leases for which an replica is expected to manage accesses.

Once the replica that manages the shard is discovered, the actual access must occur. There are several mechanisms for this that use quorums of replicas to make

decisions. Weak consistency models of access use overlapping read and write quorums of varying sizes along with eventual replication of the data [37]. Strong consistency models of access use Paxos [60] as the basis for high performance data storage [15,61] – consensus will be discussed in detail in the next section. To further increase write throughput, accesses append commands to a distributed log that are applied asynchronously to the underlying data store, so long as the command is committed to the log, it is guaranteed to be written to storage [62–64].

Finally, data must be written to stable storage, usually on clusters of disks that are also distributed so as to prevent data loss in the likely event that a disk fails. Many geo-replicated data systems also use a distributed file system for underlying storage. BigTable stores its data on GFS [65], f4 [66] on top of HDFS [67], an open source implementation of GFS, and Spanner stores its data on Colossus [68], the next generation of GFS. Other distributed databases such as BTrDB [69] use the Ceph [70] file system for data replication and because of requirements for location-fault tolerance, it is becoming increasingly rare that hardware based schemes such as RAID [71] are used to ensure data durability.

The description above and Figure 2.1 are a useful blueprint for designing large scale geo-replicated systems and generalizes many of the themes and attributes of a wide variety of systems. The problem with this blueprint is that it imposes a multi-process system architecture; replicas are coordinated by master processes and lock services, and then store data on distributed file systems, yet more processes with independent coordination. Multiprocess systems then must be further coordinated so that the health and status of each process must be known, leading to the use

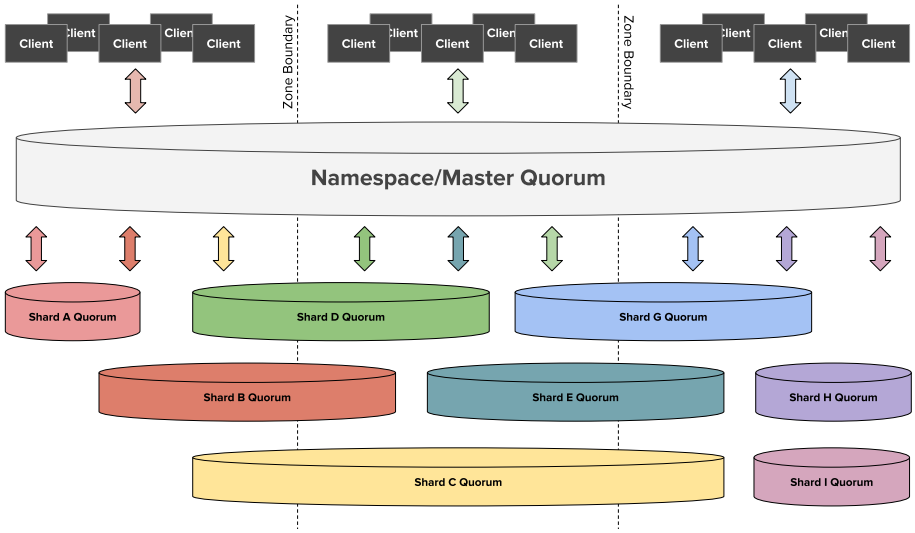


Figure 2.2: Figure 2.1 can be simplified to a single consensus architecture with a top-level consensus quorum making namespace decisions and directing requests to per-object quorums that are replicated within a single data center or across regions or zones.

of monitoring and management tools like Ambari [72] and Zookeeper [73]. With so many layers of coordination, it becomes impossible to reason about consistency and data locality, and such systems become very difficult to deploy without excellent systems administration.

We propose that the complexity of this blueprint can be simplified instead to a multiple consensus process model as shown in Figure 2.2. This model does not eliminate the components described in the blueprint, but rather consolidates them into two primary coordination activities: coordinating the namespace and coordinating accesses to and storage of objects. In this model, a single geo-replicated quorum manages the global namespace – the primary master process. Multiple independent subquorums manage accesses to individual objects, replicated solely

within a single datacenter, replicated across zones, or replicated across the wide area.

This simplification makes it clear that the core of a fault-tolerant, geo-replicated distributed system is effective distributed consensus that can scale to multiple replicas across many regions and can survive failures that may occur in wide area systems. In the next section, we will build upon this idea and describe an overview of our proposed architecture along with consistency and failure requirements.

## 2.2 System Architecture

We propose a consistency-centric approach to designing distributed data stores, centered on geographically distributed consensus. Modern software is developed with international audiences in mind from the outset and requires data services that span oceans, continents, and political regions. Existing large-scale database and file systems were purpose built for gigantic applications created by large Internet companies and include specializations for data-center level computer engineering. These specializations resulted in complex coordination divided between levels to manage transactions, namespace allocation, accesses, and storage. To ensure a wider audience of software developers can correctly reason about consistency across the wide area a single, global consistency model is required.

Geographically distributed consensus is not sufficient, however, as system environments are migrating outside of the data center. The next generation of geo-distributed systems will require edge replicas to support high-throughput writes

from sensor networks deployed on the energy grid, traffic coordination networks, and the Internet of Things. A consistency-centric approach requires therefore that both strong consistency and high availability replicas are federated into a single model of consistency. Our architecture therefore leverages a hierarchical consensus model to provide strong consistency across regions, centralized by data center along with a federated consistency model for a fog of edge devices surrounding data centers to support heterogenous network environments.

In this section we will first describe a consistency model that informs our architectural decisions. Next, we will describe the requirements for distributed systems that our architecture addresses. Finally, we will provide an overview of our planet-scale architecture that serves as the foundation for this dissertation.

### 2.2.1 Consistency and Consensus

Our consistency model is a *data-centric* model, as opposed to a *client-centric* model [74]. Client-centric models view the system as a black box and consistency is described as guarantees made to processes or applications that interact with the system such as "read your writes" or "write follows read" [75]. Data-centric consistency on the other hand is concerned about the ordering of operations applied to a replica and generally considers the problem of how those operations relate to each other in a per-replica, append-only log.

Data-centric consistency can be reasoned about by considering a log-based model of consistency. Replicas in a distributed system can be viewed as independent

state machines that apply commands in response to client requests or messages from other replicas [76]. Each command is appended to a log that records a time-ordered sequence of operations such that from the same starting state, every time the log is replayed the replica will reach the same ending state. Two replicas are locally consistent (consistent with each other) if their logs are able to bring them to identical states. Global consistency requires all replicas logs express a single, abstract ordering that brings the entire system to identical states.

Neither local nor global consistency requires replica logs to be identical, only that a log, when applied, leads to the same state. Consistency guarantees can therefore be described by specifying how the logs of two replicas or all logs in a system are allowed to vary. These variations can generally be described along two dimensions: staleness and ordering.

1. *Staleness* refers to how far behind the latest global state a local log is and can be expressed by the visibility latency of distributing a command to all replicas or simply by how many updates the log is behind by.

2. *Ordering* refers to how closely individual logs adhere to an global chronological ordering of commands. A strict ordering requires all logs or some prefix of the log to be identical, whereas weaker ordering allows some divergence in order updates applied to the log.

Most data-centric consistency models refer to the strictness of ordering guarantees and the method by which updates are applied to the state of the replica [77]. The least strict model, weak consistency (WC) makes no guarantees whatsoever

about the relationship of local and remote writes and requires no coordination. Eventual consistency (EC) is primarily concerned with the final state of all logs given some period of quiescence that allows the system to converge [78]. Though two logs may have an entirely different ordering of commands and not all commands may be present in all logs, EC guarantees that all replicas will achieve the same state, eventually. Causal consistency (CC) ensures that a log is as up to date as all other logs with respect to a subset of dependencies [79, 80]. Sequential consistency (SC) is a strong consistency model that requires all replicas have the same exact log ordering on a per-object or multi-object basis, but does not make guarantees about staleness [81, 82]. Finally, linearizablity (LIN), the strongest form of consistency, requires that clients see a single, externalizable log no matter which part of the system they access [83].

Consensus algorithms [60, 84–92] are used to coordinate the logs of replicas to provide strong consistency in a distributed system. Consensus requires two phases to ensure a command is correctly committed to a majority of replica logs. The first phase, *PREPARE*, allows a replica to nominate a slot in the log for a specific command. If the majority of the replicas agree to allow the replica that slot, the second phase, *ACCEPT*, allows a majority of replicas to agree that they have placed a specific command in specified log slot. At the cost of multiple coordination messages per access, consensus guarantees that all replicas will always have identical logs.

Because enforcing log ordering requires increased coordination between replicas, there is a trade-off between ordering strictness and staleness that often defines the choice of consistency model used in a distributed system. Coordination adds de-

pendencies to accesses that introduces latency when responding to clients and total failure if part of the system is not available [16]. Eventually consistent models reduce coordination and susceptibility to failure by creating relaxing quorum membership and using asynchronous synchronization. By relaxing ordering strictness, the system is able to respond more quickly but the reduction in coordination causes staleness, which is the root of all observed inconsistencies in the system [93, 94]. Staleness is entirely dependent on latency, therefore, in a data-center context, eventual consistency has been considered consistent enough. In a geo-replicated context, however, the requirements for data systems change as the physical properties of networks become more apparent.

### 2.2.2 Requirements for Data Systems

We contend that *consistency depends on the network environment.* A network with instantaneous and perfectly reliable communications would never be inconsistent because all updates could be applied simultaneously with no latency. Real world networks have to contend with physical systems and distances that create meaningful delays when coordinating messages. Eventually consistent systems depend on the speed at which updates are disseminated through the network – the slower the dissemination, the more likely that an inconsistency is observed. Strong consistency systems implemented with consensus are provably correct but will fail to make progress as network conditions deteriorate. In a geo-replicated system, consistency challenges are even greater because latencies are larger and outages more

widespread. Most proposed geo-replicated systems [47, 95–99] therefore attempt to find some balance of consistency models, trading off between the types of expected failures. In this section we will briefly describe our expectations for network conditions and the requirements for our data system.

In large systems with thousands of replicas and millions of clients, failure is common and should be expected [13]. Disk failure is the most destructive form of failure in a system because it leads to permanent data loss and can only be managed through redundancy. Replica failure either due to hardware failure or power loss, though temporary, reduces the total availability of the system. Homogeneity in both disks and replicas can also lead to correlated failures, causing an extremely destructive cascading effect [66].

In addition to node failures, communication failure must also be resolved. We assume a reliable network protocol that buffers messages and ensures delivery if a replica can be communicated with, messages are not dropped so long as the recipient is online. We therefore treat network failures as partitions such that replicas cannot communicate with some subset of its peers. In the case of either replica or network failure, once replicas can communicate and are back online, they must be able to gracefully rejoin the system.

In a geo-replicated context, large latencies are not primary issue, rather variability in expected latencies are. Access patterns are typically location-dependent and correlated with respect to time (e.g. there are more accesses during daylight business hours). There is a known physical limit to message traffic and with deterministic latency a network could be constructed to efficiently and correctly propa-

gate data around the system. Unfortunately, because both partitions and latency are variable, systems must be designed to be fully connected to all areas of the network.

In the face of failure, the primary requirements for a geo-replicated system are therefore durability, fault-tolerance, heterogeneity, and adaptability. Durability normally considers three disk replication to ensure that 2 failures do not lead to data loss. In a geo-replicated context, regions provide robustness in the case of catastrophic failures, e.g. a natural disaster that causes wide-spread power failures [100]. Therefore durability and fault tolerance require not just disk replication but also zone and region replication. Heterogeneity prevents both cascading, correlated failure, but also allows many different types of systems to participate in the network. Finally adaptability allows the system to respond to changes in the network environment, both in terms of outages and user access patterns. An adaptable system will also be able to dynamically add and remove nodes and scale with an increased number of regions. With these requirements in mind, in the next section we describe our proposed architecture.

### 2.2.3 A Planetary Scale Architecture

We envision a consistency-centric, planetary-scale distributed system as a two tier architecture shown in Figure 2.3. The first tier resides inside of a data center environment and relies on high-speed backbone connections and high performance machines to implement geo-distributed consensus using the hierarchical consensus

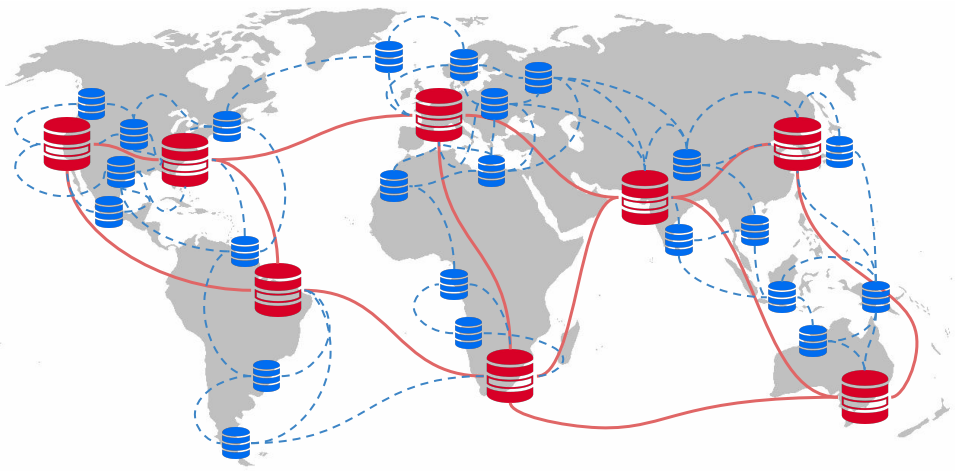

Figure 2.3: A global architecture composed of a core backbone of hierarchical consensus replication (red) and a fog of heterogenous, federated consistency replicas (blue).

protocol, which we describe in Chapter 3. The second tier is a highly-available network of edge replicas that disseminate updates in the wide area between data centers using a federated consistency model, which we will describe in Chapter 4. Such a large system requires machine learning mechanisms to monitor and adapt behavior according to changing network conditions, maximizing the consistency of the entire system, which we will describe in Chapter 6.

Modern software applications require a strong consistency model that is region-agnostic. Hierarchical consensus provides that consistency model by unifying coordination into a single-process model rather than having multiple, independent processes all coordinating accesses. Hierarchical consensus ensures that there is an intersection between namespace coordination and access coordination so that there is a provably strong relationship between the participation of all replicas in

consensus across the globe. This relationship ensures that a general audience of developers can reason about consistency, geolocate data, and deploy systems without the complexity of most systems.

The next generation of systems will also require high-throughput writes from a mobile, heterogenous network. Rather than relying on the centralizing effect of the cloud, we also propose to augment our system with a decentralized fog of highly available replicas. These replicas use traditional eventually consistent systems to ensure high-availability for writes at the cost of a high likelihood of stale reads. We propose that this outer edge layer is not independent of the centralized applications, but rather we propose a federation of consistency models that increases global consistency. Federated consistency allows replicas to choose at which consistency level they participate in the system, creating a continuous, hybrid consistency across independent objects.

The base application we have constructed is a key/value store as described in Chapter 5. Keys serve as the basis for sharding in our system and allow us to generically apply dependency relationships depending on the application. Key/value stores can be seen as the underlying storage for databases, but we target two other applications: a distributed ledger and a file system. Distributed ledgers have recently grown in popularity thanks to decentralized blockchain protocols [101]. Hierarchical consensus can be used to quickly export a per-object or multi-object distributed log. Key/value stores can also be used for underlying file systems [102]. Many high-performance distributed databases rely on an underlying replicated file system [36, 40, 41, 50, 69]. We believe that a planet-scale file system will therefore provide the

best platform to construct a myriad of services that are themselves planet-scale.

## 2.3 Conclusion

In this chapter we have presented the challenges, motivations, and background of today's geo-replicated data systems. Modern software systems are now developed with international audiences in mind, largely due to the success of large Internet companies that provide cloud infrastructure around the world. As the demand for global-scale software has grown, so to has the need for geo-replicated services, however, while cloud providers have provided access to provisioned large scale databases, these database have been specialized and optimized for the applications they were built for, not a general audience. The result is that software developers have to deeply consider consistency and localization semantics at the application level, which leads to confusion.

The challenge is that the current generation of geo-distributed systems rely on a pristine data-center environment, able to support a multi-process architecture on high-performance machines and networks. Multi-process architectures have multiple levels of coordination and replication, making it extremely difficult to reason about the consistency model. Moreover, the next generation of geo-distributed system will not reside in data-centers, but in more variable, mobile network environments. To accommodate both of these trends, we have proposed a consistency-centric architecture for planet-scale systems.

The primary challenges for a planet-scale systems are their scale and the vari-

ability of the connections between participants. Our architecture places primary importance on durability, fault-tolerance, heterogeneity, and adaptability by specifying two federated tiers of access. The first tier, inside of data centers, uses hierarchical consensus to provide strong geo-replicated consistency as well as catastrophic failure tolerance by replicating data cross zones and regions. The second tier, at the edge in the mobile network federates a highly available system model with a strong consistency model to provide stronger global guarantees. Finally, our system self-organizes by monitoring access patterns and the network environment, adapting to change to provide resilience over time.

In the next chapter, we will explore the core component of this dissertation: hierarchical consensus.

# Chapter 3: Hierarchical Consensus

The backbone of our planetary scale data system is *hierarchical consensus* [103]. Hierarchical consensus provides a strong consistency foundation that totally orders critical accesses and arbitrates the eventual consistency layer in the fog, which raises the overall consistency of the system. To be effective, an externalizable view of consistency ordering must be available to the entire system. This means that strong consistency must be provided across geographic links rather than provided as localized, independent decision making with periodic synchronization. The problem that hierarchical consensus is therefore designed to solve is that of geographically distributed consensus.

Solutions to geo-distributed consensus primarily focus on providing high throughput, low latency, fault tolerance, and durability. Current approaches [43, 87, 89, 104–106] usually assume few replicas, each centrally located on a highly available, powerful, and reliable host. These assumptions are justified by the environments in which they run: highly curated environments of individual data centers connected by dedicated networks. Although replicas in these environments may participate in global consensus, our architecture requires us to accommodate replicas with heterogenous capabilities and usage modalities. Widely distributed replicas might have neither

high bandwidth nor low latency and might suffer partitions of varying durations. Such systems of replicas might also be dynamic in membership, in relative locations, and have relative workloads. Most importantly, to provide a backbone for a planetary scale data system, the consistency backbone must scale to include potentially thousands of replicas around the world.

As a result, straightforward approaches of running variants of Paxos [60, 84], ePaxos [87], or Raft [92] across the wide area, even for individual objects will perform poorly for several reasons. First, distance (in network connectivity) between consensus replicas and the most active replicas decrease the performance of the entire system, consensus is only as fast as the final vote required to make a decision, even when making thrifty requests. Second, network partitions are common, which cause consensus algorithms to fail-stop [107] if they cannot receive a majority, a criticism that is often used to justify eventual consistency systems for high availability. Finally, the fault tolerance of small quorum algorithms can be disrupted by a few unreliable hosts and given the scale of the system and the heterogenous nature of replicas, the likelihood of individual failure is so high so as to be considered inevitable.

We propose another approach to building large consensus systems. Rather than relying on a few replicas to provide consensus to many clients, we propose to run a consensus protocol across replicas running at or near all of these locations. The key insight is that large problem spaces can often be partitioned into mostly disjoint sets of activity without violating consistency. We exploit this decomposition property by making our consensus protocol hierarchical and individual consensus

groups fast by ensuring they are small. We exploit locality by building subquorums from colocated replicas, and locating subquorums near clients they serve.

In this chapter we describe hierarchical consensus, a tiered consensus structure that allows high throughput, localization, agility, and linearizable access to a shared namespace. We show how to use *delegation* (§ 3.2.3) to build large consensus groups that retain their fault tolerance properties while performing like small groups. We describe the use of *fuzzy epoch transitions* (§ 3.2.5) to allow global reconfigurations across multiple consensus groups without forcing them into lockstep. Finally, we describe how we reason about consistency by describing the structure of *grid consistency* (§ 3.3.1).

## 3.1 Overview

Hierarchical Consensus (HC) is a leader-oriented implementation and extension of Vertical Paxos [108–110] designed to scale to hundreds of nodes geo-replicated around the world. Vertical Paxos divides consensus decisions both horizontally, as sequences of consensus instances, and vertically as individual consensus decisions are made. Spanner [41], MDCC [42], and Calvin [43], can all be thought of as implementations of Vertical Paxos. These systems shard the namespace of the objects they manage into individual consensus groups (the horizontal division) each independently reaching consensus about accesses (the vertical division). A general implementation of Vertical Paxos therefore requires multiple independent quorums, one to manage the namespace and keep track of the health of all replicas in the

system, and many other subquorums to independently manage tablets or objects.

In a geo-replicated context, there are two problems with this scheme. First, sharding does not allow for inter-object dependence (in the horizontal division) without relying on coordination from the management quorum. Second, the management quorum must be geo-replicated and able to scale to handle monitoring of the entire system. In both cases, the management quorum becomes a bottleneck and current solutions such as batching decisions or using specialized hardware for extremely accurate timestamps are outside of the scope of the safety properties provided by Vertical Paxos. The challenge is therefore in constructing a multi-group coordination protocol that configures and mediates per-object quorums with the same level of consistency and fault tolerance as the entire system.

Hierarchical consensus organizes *all* participating replicas into a single root quorum that manages the namespace across all regions as shown in Figure 3.1. The root quorum guarantees correctness and failure-free operation by pivoting the overall system through two primary functions. First, the root quorum reconfigures its membership into subquorums, reserving extra members as hot-spares if needed. Second, the root quorum adjusts the mapping of the object namespace to the underlying partitions, which subquorums are responsible for managing. These adjustments adapt the system to replica failures, system membership changes, varying access patterns, and ensure that related objects are coordinated together. Much of the system's complexity comes from handshaking between the root quorum and the lower-level subquorums during reconfigurations.

These handshakes are made easier, and much more efficient, by using *fuzzy*

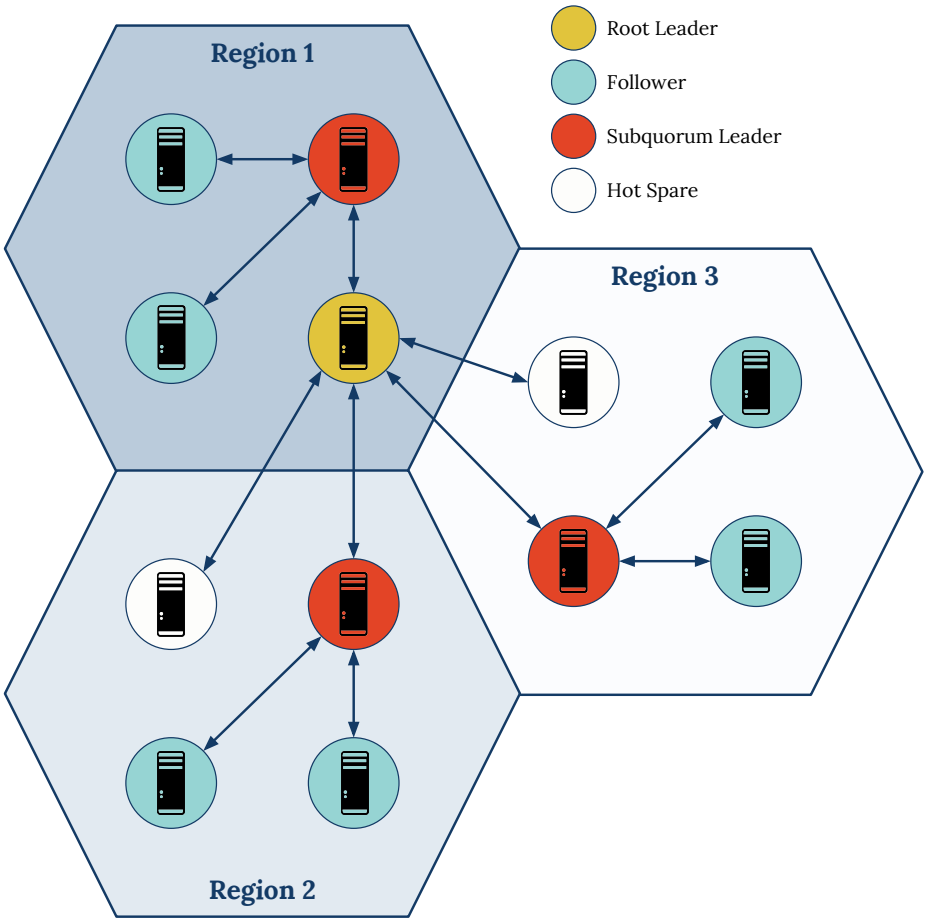


Figure 3.1: A simple example of an HC network composed of 12 replicas with size 3 subquorums. Each region hosts its own subquorum and subquorum leader, while the subquorum leaders delegate their votes to the root quorum, whose leader is found in region 1. This system also has 2 hot spares that can be used to quickly reconfigure subquorums that experience failures. The hot spares can either delegate their vote, or participate directly in the root quorum.

*transitions*. Fuzzy transitions allow individual subquorum to move through reconfiguration at their own pace, allowing portions of the system to transition to decisions made by the root quorum before others. Given our heterogenous, wide-area environment, forcing the entire system to transition to new configurations in lockstep would be unacceptably slow. Fuzzy transitions also ensure that there is no dedicated shard-master that has to synchronize all namespace allocations: at the cost of possibly multiple redirections, clients can be redirected by any member of the root quorum to replicas who should be participating in consensus decisions for the requested objects. Fuzzy transitions ensure that root quorum decisions need not be timely since those decisions do not disrupt accesses of clients.

Though root quorum decisions are rare with respect to the throughput of accesses inside the entire system, they still do require the participation of all members of the system, which could lead to extremely large quorum sizes, and therefore extremely slow consensus operations that may be extremely sensitive to partitions. Because all subquorums make disjoint decisions and because all members of the system are part of the root quorum, we propose a safe relaxation of the participation requirements for the root quorum such that subquorum followers can *delegate* their root quorum votes to their leader. Delegation ensures that only a few replicas participate in most root quorum decisions, though decisions are made for the entire system.

In brief, the resulting system is local, in that replicas serving clients can be located near them. The system is fast because individual operations are served by small groups of replicas, regardless of the total size of the system. The system

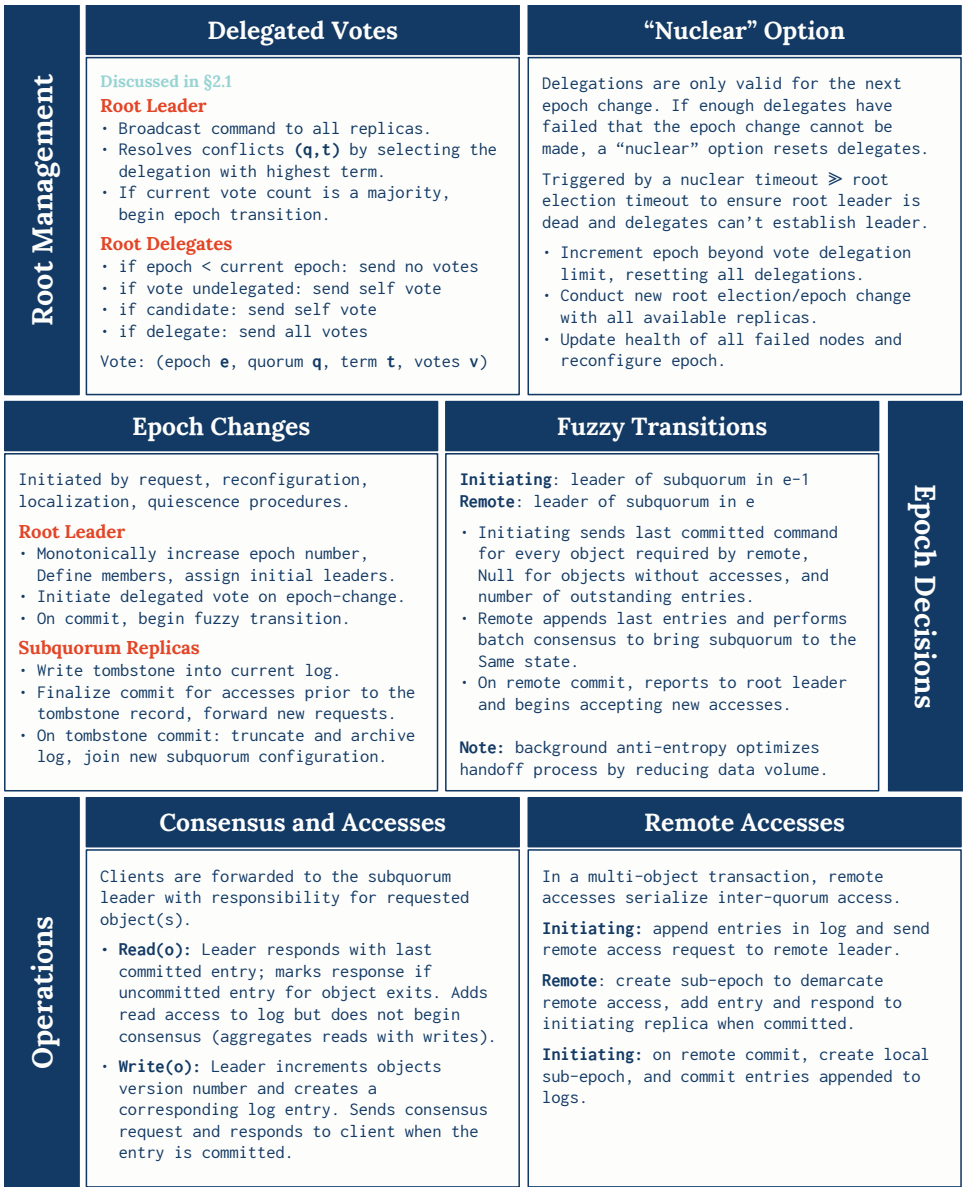


Figure 3.2: A condensed summary of the hierarchical consensus protocol. Operations are described top-to-bottom where the top level is root quorum operations, the bottom is subquorum operations, and the middle is transition and intersection.

is nimble in that it can dynamically reconfigure the number, membership, and responsibilities of subquorums in response to failures, phase changes in the driving applications, or mobility among the member replicas. Finally, the system is consistent, supporting the strongest form of per-object consistency without relying on special-purpose hardware [41, 111–114].

A complete summary of hierarchical consensus is described in Figure 3.2.

## 3.2 Consensus

The canonical distributed consensus used by systems today is Paxos [60, 84]. Paxos is provably safe and designed to make progress even when a portion of the system fails. As described in § 2.2.1, consensus operations maintain a single, ordered log of operations that consistently change the state of the replica. Raft [92] was designed not to improve performance, but to increase understanding of consensus behavior to better allow efficient implementations. HC uses Raft as a building block, so we describe the relevant portions of Raft at a high level, referring the reader to the original paper for complete details. Though we chose to base our protocol on Raft, a similar approach could be used to modify Paxos or one of its variants into a hierarchical structure.

Consensus protocols have two phases: leader *election* (also known as `PREPARE`) and operations *commit* (also known as `ACCEPT`). Raft is a strong-leader consensus protocol, a common optimization of Paxos variants called multi-Paxos [85, 86, 89, 115]. Multi-Paxos allows the election phase to be elided while a leader remains avail-

able, therefore the protocol requires only a single communication round to commit an operation in the common case.

Raft uses timeouts to trigger phase changes and provide fault tolerance. Crucially, it relies on timeouts only to provide progress, not safety. New elections occur when another replica in the quorum times out waiting for communication from the leader. Such a replica increments its *term* until it is greater than the existing leader, and announces its candidacy by sending a `VoteRequest`. Other replicas vote for the candidate if they have not seen a competing candidate with a larger term and become followers, waiting for entries from the leader to be appended to its log.

During regular operation, clients send requests to the leader, which broadcasts `AppendEntries` messages carrying operations to all followers. Term-invariants guarantee safety, followers will only accept an append entry request from a leader with a term as high or higher than the follower's term. Additionally, followers will not append an entry to a log unless the leader can prove that the follower's log is as up to date as its own, determined by the index and term of the leader's previous entry. An operation is *committed* and can be executed when the leader receives acknowledgments of the `AppendEntries` message from more than half the replicas (including itself).

HC implements an adapted Raft consensus algorithm at both the root and subquorums. We chose Raft both for its understandability in implementation and to easily describe operations in a leader-oriented context – for example delegation is more understandable in the context of voting to elect a leader. We describe differences in our Raft implementation from the canonical implementation in Chapter 5.

### 3.2.1 Terminology and Assumptions

Throughout the rest of this chapter we use the term *root quorum* to refer to the upper, namespace-mapping and configuration-management tier of HC, and *subquorum* to describe a group of replicas (called *peers*) participating in consensus decisions for a section of the namespace. The root quorum shepherds subquorums through *epochs*, each with potentially different mappings of the namespace and replicas to subquorums. An epoch corresponds to a single commit phase of the root quorum. We use the term Raft only when describing details particular to our current use of Raft as the underlying consensus algorithm. We refer to the two phases of the base consensus protocol as the *election phase* and the *commit phase*. We use the term *vote* as a general term to describe positive responses in either phase.

Epoch $x$ is denoted $e_x$ when referring to the numeric counter and $Q_x$ when referring to a configuration of subquorums. Subquorum $i$ of epoch $e_x$ is represented as $q_{i,x}$, or just $q_i$ when the epoch is obvious, e.g. $q_i \in Q_x$. The namespace is divided into *tags*, disjoint subsets of the namespace. It is equivalent to refer to $q_{i,x}$ as a tag since every subquorum manages at least one tag so that $Q_x$ covers the entire namespace. However, for specificity we may refer to $t_{i,x}$ and $t_i \in T_x$ when referring to objects rather than the subquorums that manage them. A system is composed of $R$ replicas, an individual replica may be designated $r_k$ to distinguish an individual replica from a subquorum.

We assume faults are fail-stop [107] rather than Byzantine [116]. We do not assume that either replica hosts or networks are homogeneous, nor do we assume

freedom from partitions and other network faults.

### 3.2.2 Root Consensus

Hierarchical consensus is a leader-oriented protocol that organizes replicas into two tiers of quorums, each responsible for fundamentally different decisions (Figure 3.3). The lower tier consists of multiple independent *subquorums*, each committing operations to local shared logs. The upper, *root quorum*, consists of subquorum peers, usually their leaders, delegated to represent the subquorum and hot spares in root elections and commits. Hierarchical consensus's main function is to export a linearizable abstraction of shared accesses to some underlying substrate, such as a distributed object store or file system. We assume that nodes hosting object stores, applications, and HC are frequently co-located across the wide area.

The root quorum's primary responsibilities are mapping replicas to individual subquorums and mapping subquorums to tags within the namespace. Each such map defines a distinct epoch, $e_x$, a monotonically increasing representation of the term of the configuration of subquorums and tags, $Q_x$. The root quorum is effectively a consensus group consisting of subquorum leaders. Somewhat like subquorums, the effective membership of the root quorum is not defined by the quorum itself, but in this case by leader election or peer delegations in the lower tier. While the root quorum is composed of all replicas in the system, only this subset of replicas actively participates in root quorum decision making.

The root quorum partitions (shards) the namespace across multiple subquo-

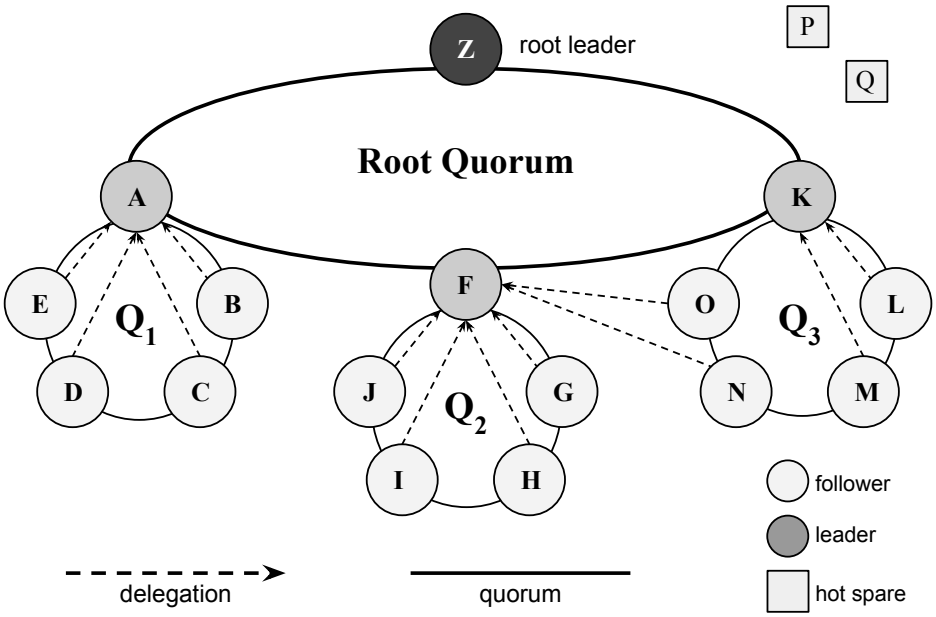


Figure 3.3: The root quorum coordinates all replicas in the system including hot spares, though active participation is only by delegated representatives of subquorums, which do not necessarily have to be leaders of the subquorum, though this is most typical. Subquorums are configured by root quorum decisions which determine epochs of operation. Each subquorum handles accesses to its own independent portion of the namespace.

rums, each with a disjoint portion as its scope. The namespace is decomposed into a set of tags, $T$ where each tag $t_i$ is a disjoint subset of the namespace. Tags are mapped to subquorums in each epoch, $Q_x \mapsto T_x$ such that $\forall t \in T_x \; \exists! q_{i,x} \mapsto t$. The intent of subquorum localization is ensure that the *domain* of a client, the portion of the namespace it accesses, is entirely within the scope of a local, or nearby, subquorum. If true across the entire system, each client interacts with only one subquorum, and subquorums do not interact at all during execution of a single epoch. This *siloing* of client accesses simplifies implementation of strong consistency guarantees and allows better performance.

### 3.2.3 Delegation

The root quorum's membership is, at least logically, the set of all system replicas, at all times. However, running consensus elections across large systems is inefficient in the best of cases, and prohibitively slow in a geo-replicated environment. Root quorum decision-making is kept tractable by having replicas *delegate* their votes, usually to their leaders, for a finite duration of epochs. With leader delegation, the root membership effectively consists of the set of subquorum leaders. Each leader votes with a count describing its own and peer votes from its subquorum and from hot spares that have delegated to it.

Fault tolerance scales with increasing system size and consensus leadership is not intended to be stable. A quorum leader is elected to indefinitely assign log entries to slots (access operations for subquorums, epoch configurations for the root

quorum). If the leader fails, then so long as the quorum has enough online peers, they can elect a new leader, and when the leader comes back online, it rejoins the quorum as a follower. The larger the size of the quorum, the more failures it is able to tolerate. This means that there might be multiple subquorum leaders in a single epoch as shown in Figure 3.4.

Consider an alternative leader-based approach where root quorum membership is defined as the current set of subquorum leaders. Both delegation and the leader approach have clear advantages in performance and flexibility over direct votes of the entire system. However, the leader approach causes the root quorum to become unstable as its membership changes during partitions or subquorum elections. These changes would require heavyweight *joint consensus* decisions in the root quorum for correctness in Raft-like protocols [92]. By delegating at the root level, we introduce the possibility that a delegate fails, removing many root votes, to ensure root quorum stability (we address this possibility in § 3.4.2).

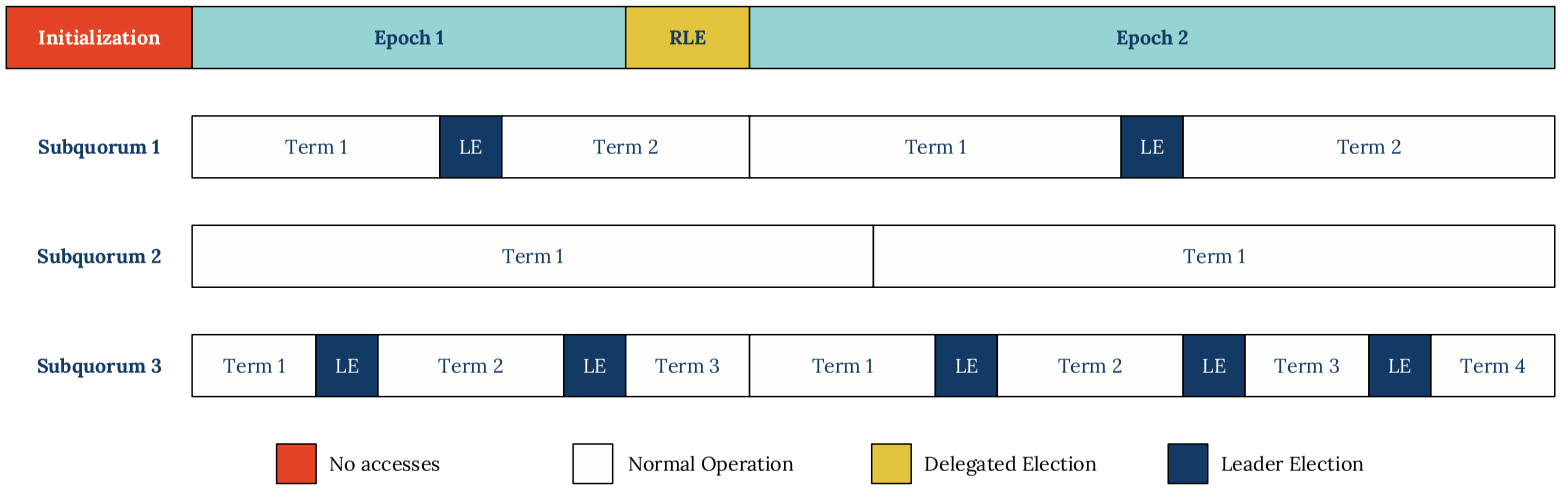


Figure 3.4: The relationship of subquorum leadership (terms) to epochs is shown as follows. Three subquorums participate in HC in variable network conditions. In $e_1$, $q_{1,1}$ and $q_{3,1}$ experience outages causing them to elect new subquorum leaders while $q_{2,1}$ remains stable. After a root election transitions the system from $e_1 \rightarrow e_2$ (with a fuzzy transition from $q_{2,1} \rightarrow q_{2,2}$). The configuration $Q_2$ creates three totally new subquorums whose terms and leader election restarts.

Delegation ensures that root quorum membership is always the entire system and remains unchanged over subquorum leader elections and even reconfiguration. Delegation is essentially a way to optimistically shortcut contacting every replica for each decision. Subquorum repartitioning merely implies that a given replica's vote might need to be delegated to a different leader. To ensure that delegation happens correctly and without requiring coordination, we simply allow a replica to directly designate another replica as its delegate until some future epoch is reached. Replicas may only delegate their vote once per epoch and replicas are not required to delegate their vote. To simplify this process, during configuration of subquorums by the root quorum, the root leader provides delegate hints, e.g. those replicas that have been stable members of the root quorum without partitions. When replicas receive their configuration they can use these hints to delegate their vote to the closest nearby delegate if not already delegated for the epoch. If no hints are provided, then replica followers generally delegate their vote to the term 1 leader and hot spares to the closest subquorum leader.

Delegation does add one complication: the root quorum leader must know all vote delegations to request votes when committing epoch changes. We deal with this issue, as well as the requirement for a nuclear option (§ 3.4.2), by simplifying our protocol. Instead of sending vote requests just to subquorum leaders, **the root quorum leader sends vote requests to all system replicas.** This is true even for *hot spares*, which are not currently in any subquorum. Delegates reply with the unique ids of the replicas they represent so that root consensus decisions are still made using a majority of all system replicas.

This is correct because vote requests now reach all replicas, and because replicas whose votes have been delegated merely ignore the request. We argue that it is also efficient, as a commit's efficiency depends only on receipt of a majority of the votes. Large consensus groups are generally slow (see Figure 3.9) not just because of communication latency, but because large groups in a heterogeneous setting are more likely to include replicas on very slow hosts or networks. In the usual case for our protocol, the root leader still only needs to wait for votes from the subquorum leaders. Leaders are generally those that respond more quickly to timeouts, so the speed of root quorum operations is unchanged.

### 3.2.4 Epoch Transitions

Every epoch represents a new configuration of the system as designated by the root leader. Changing configuration ensures that the system is both dynamic, responding both to failures and changing usage patterns, and to minimize coordination by colocating related objects. An epoch change is initiated by the root leader in response to one of several events, including:

- a namespace repartition request from a subquorum leader
- notification of join requests by new replicas
- notification of failed replicas
- changing network conditions that suggest re-assignment of replicas
- manual reconfigurations, e.g. to localize data

The root leader transitions to a new epoch through the normal commit phase in the root quorum. The command proposed by the leader is an enumeration of the new subquorum partition, namespace partition, and assignment of namespace portions to specific subquorums. The announcement may also include initial leaders for each subquorum, with the usual rules for leader election applying otherwise, or if the assigned leader is unresponsive. Upon commit, the operation serves as an *announcement* to subquorum leaders. Subquorum leaders repeat the announcement locally, disseminating full knowledge of the new system configuration, and eventually transition to the new epoch by committing an `epoch-change` operation locally.

The epoch change is lightweight for subquorums that are not directly affected by the underlying re-configuration. If a subquorum is being changed or dissolved, however, the *epoch-change* commitment becomes a tombstone written to the logs of all local replicas. No further operations will be committed by that version of the subgroup, and the local shared log is archived and then truncated. Truncation is necessary to guarantee a consistent view of the log within a subquorum, as peers may have been part of different subquorums, and thus have different logs, during the last epoch. Replicas then begin participating in their new subquorum instantiation. In the common case where a subquorum's membership remains unchanged across the transition, an `epoch-change` may still require additional mechanism because of changes in namespace responsibility.

### 3.2.5 Fuzzy Handshakes

Epoch handshakes are required whenever the namespace-to-subquorum mapping changes across an epoch boundary. HC separates epoch transition announcements in the root quorum from implementation in subquorums. Epoch transitions are termed *fuzzy* because subquorums need not all transition synchronously. There are many reasons why a subquorum might be slow. Communication delays and partitions might delay notification. Temporary failures might block local commits. A subquorum might also delay transitioning to allow a local burst of activity to cease such as currently running transactions [1] . Safety is guaranteed by tracking subquorum dependencies across the epoch boundary.

The most complex portion of the HC protocol is in handling data-related issues at epoch transitions. Transitions may cause tags to be transferred from one subquorum to another, forcing the new leader to load state remotely to serve object requests. Transitions handshakes are augmented in three ways. First, an replica can demand-fetch an object version from any other system replica. Second, epoch handoffs contain enumerations of all current object versions, though not the data itself. Knowing an object's current version gives the new handler of a tag the ability to demand fetch an object that is not yet present locally. Finally, handshakes start immediate fetches of the in-core version cache from the leader of the tag's subquorum in the old epoch to the leader in the new.

Figure 3.5 shows an epoch transition where the scopes of $q_i$, $q_j$, and $q_k$ change

[1] The HC implementation discussed in this chapter does not currently support transactions.

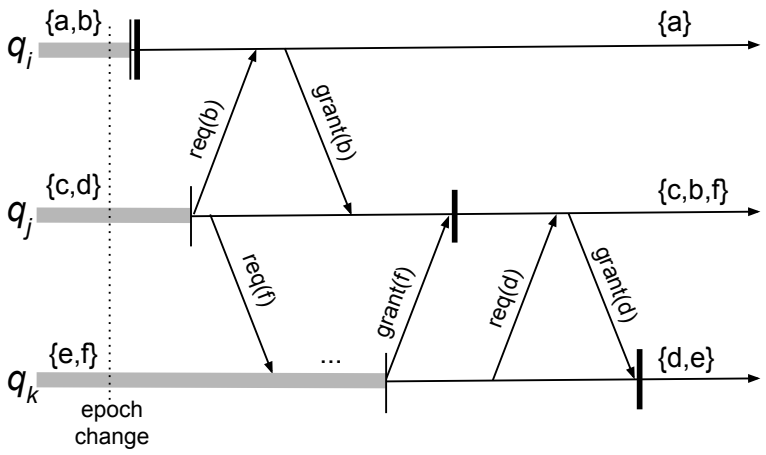


Figure 3.5: Readiness to transition to the new epoch is marked by a thin vertical bar; actual transition is the thick vertical bar. Thick gray lines indicate operation in the previous epoch. Subquorum $q_j$ transitions from tag $c, d$ to $c, b, f$, but begins only after receiving version information from previous owners of those tags. The request to $q_k$ is only answered once $q_k$ is ready to transition as well.

across the transition as follows:

$$q_{i,x-1} = t_a, t_b \quad \longrightarrow q_{i,x} = t_a \tag{3.1}$$

$$q_{j,x-1} = t_c, t_d \quad \longrightarrow q_{j,x} = t_c, t_d, t_f \tag{3.2}$$

$$q_{k,x-1} = t_e, t_f \quad \longrightarrow q_{k,x} = t_d, t_e \tag{3.3}$$

All three subquorums learn of the epoch change at the same time, but become ready with varying delays. These delays could be because of network lags or ongoing local activity. Subquorum $q_i$ gains no new tags across the transition and moves immediately to the new epoch. Subquorum $q_j$'s readiness is slower, but then it sends requests to the owners of both the new tags it acquires in the new epoch. Though $q_i$ responds immediately, $q_k$ delays its response until locally operations conclude. Once both handshakes are received, $q_j$ moves into the new epoch, and $q_k$ later follows suit.

These bilateral handshakes allow an epoch change to be implemented incrementally, eliminating the need for lockstep synchronization across the entire system. This flexibility is key to coping with partitions and varying connectivity in the wide area. However, this piecewise transition, in combination with subquorum re-definition and configuration at epoch changes, also means that individual replicas *may be part of multiple subquorums at a time.*

This overlap is possible because replicas may be mapped to distinct subgroups from one epoch to the next. Consider $q_k$ in Figure 3.5 again. Assume the epochs shown are $e_x$ and $e_{x+1}$. A single replica, $r_a$, may be remapped from subquorum $q_{k,x}$

to subquorum $q_{i,x+1}$ across the transition. Subquorum $q_{k,x}$ is late to transition, but $q_{i,x+1}$ begins the new epoch almost immediately. Requiring $r_a$ to participate in a single subquorum at a time would potentially delay $q_{i,x+1}$'s transition and impose artificial synchronicity constraints on the system. One of the many changes we made in the base Raft protocol is to allow a replica to have multiple distinct shared logs. Smaller changes concern the mapping of requests and responses to the appropriate consensus group.

### 3.2.6 Subquorum and Client Operations

A subquorum, $q_{i,x}$, logically exists only for the duration of an epoch, $e_x$ and maps accesses to a subset of tags in $T$ such that $q_{i,x} \mapsto t_{i,x} \subset T_x$. Each subquorum elects a leader to coordinate local decisions. Fault tolerance of the subquorum is maintained in the usual way, detecting leader failures and electing new leaders from the peers. Subquorums do not, however, ever change system membership on their own. Subquorum membership is always defined in the root quorum.

Subquorum consensus is used to manage client accesses by committing object writes and designating a responding replica for object reads. Clients can generally `Get` a key (a read operation), and can `Put` values and `Del` objects (write operations). Client accesses are forwarded to the leader of the subquorum for the appropriate tag the object being accessed belongs to. Because the root quorum manages the namespace, all replicas can correctly forward a client to a member of the subquorum, at worst requiring two redirects to reach a leader. The underlying Raft semantics en-

sure that leadership changes do not result in loss of any commits. Hence, individual- or multiple-client accesses to a single subquorum are totally ordered.

All writes are committed as operations by consensus decisions, appending the operation to a log shared by all replicas. On commit, the write is applied to the underlying storage asynchronously, so long as the write is committed, it is guaranteed to be applied in the order specified by the log. The shared logs also provide a complete version history of all distributed objects. Subquorum leaders use in-core caches to provide fast access to recently accessed objects in the local subquorums's tag. Replicas perform background anti-entropy [37, 78, 117], disseminating log updates a user-defined number of times across the system, providing both durability as well as fast transitions between configurations.

Reads are not committed with consensus decisions by default. Leaders respond to `Get` requests by replying with the latest applied (committed) value. This introduces the possibility of a stale read, e.g. that a read occurs before a committed operation is applied. To ensure a subquorum has linearizable consistency, reads would also need to be committed. Another option is to commit read-leases to a specified client, so that they are guaranteed to read a snapshot of object values. Committing multi-object leases is the basis for implementing transactions in subquorums, however they also play an important role in accessing multiple objects across subquorum boundaries.

Although we have not yet implemented transactions in our system, we have provided a mechanism for multi-object coordination in the case where objects span multiple subquorums. A *remote access* is conducted from one subquorum to another

(e.g. leader to leader) transparently from the client. Remote accesses have implications for consistency as described in § 3.3.1. For now we simply point out that all remote accesses, both reads and writes, require the subquorum to commit the operation or grant a temporary lease to the remote quorum. To optimize this process, batch commands may be used to limit the amount of cross-region communication required for remote accesses.

## 3.3 Consistency

Hierarchical consensus provides the strongest possible per-object and global consistency guarantees. Pushing all writes through subquorum commits and serving reads at leaders allows us to guarantee that per-object accesses are linearizable (Lin), which is the strongest non-transactional consistency [81, 83]. As a recap, linearizability is a combination of atomicity and timeliness guarantees about accesses to a single object. Both `reads` and `writes` must appear atomic, and also instantaneous at some time between a request and the corresponding response to a client. `Reads` must always return the latest value. This implies that reads return values are consistent *with any observed ordering*, i.e., the ordering is *externalizable* [118].

Linearizability of object accesses can be *composed*. If operations on each object are linearizable, the entire object space is also linearizable. This allows our subquorums to operate independently while providing a globally consistent abstraction. The resulting consistency model can be reasoned about as *grid consistency*

### 3.3.1 Grid Consistency

Grid consistency uses the Vertical Paxos log model to also model a global ordering of operations that ensure the state of the system is always consistent. We express total orderings as "happened-before" ($\rightarrow$) relationships [119]. The grid is defined along the horizontal by each epoch or configuration. Epochs are totally ordered, such that $e_{x-1} \rightarrow e_x$, determined by the root log's consensus operations. This implies that any access in $q_{i,x-1} \rightarrow q_{j,x}$, which is guaranteed by the tombstone and hand-off operation during epoch transition. The grid is defined vertically by logs of all $q_i \in Q_x$. Because subquorums operate independently with the exception of remote accesses, operations in each log can be applied concurrently. Said another way, all subquorum logs within an epoch implement a fuzzy log [120]. In this section we show how remote accesses and independent subquorums in a single epoch create a linearizable total ordering via composability and in the following section we describe how to extract a sequentially consistent global log.

Figure 3.6 shows a system with subquorums $q_i$ and $q_j$, each of which performs a pair of writes. Dotted lines show one possible event ordering for replicas $q_i$ (responsible for objects $a$ and $b$), and $q_j$ ($c$ and $d$). Without cross-subquorum reads or writes, ordering either subquorums's operations first creates a SC total ordering: $q_i \rightarrow q_j$ implies $w_{i,1} \rightarrow w_{i,3} \rightarrow w_{j,1} \rightarrow w_{j,3}$, for example. If the epoch has not been concluded, then the best the system can guarantee is sequential consistency, we can only guarantee that $w_{j,1} \rightarrow w_{j,3}$ and that $w_{i,1} \rightarrow w_{i,3}$. Once the epoch is concluded, however, there is no other ordering other than the one expressed by the

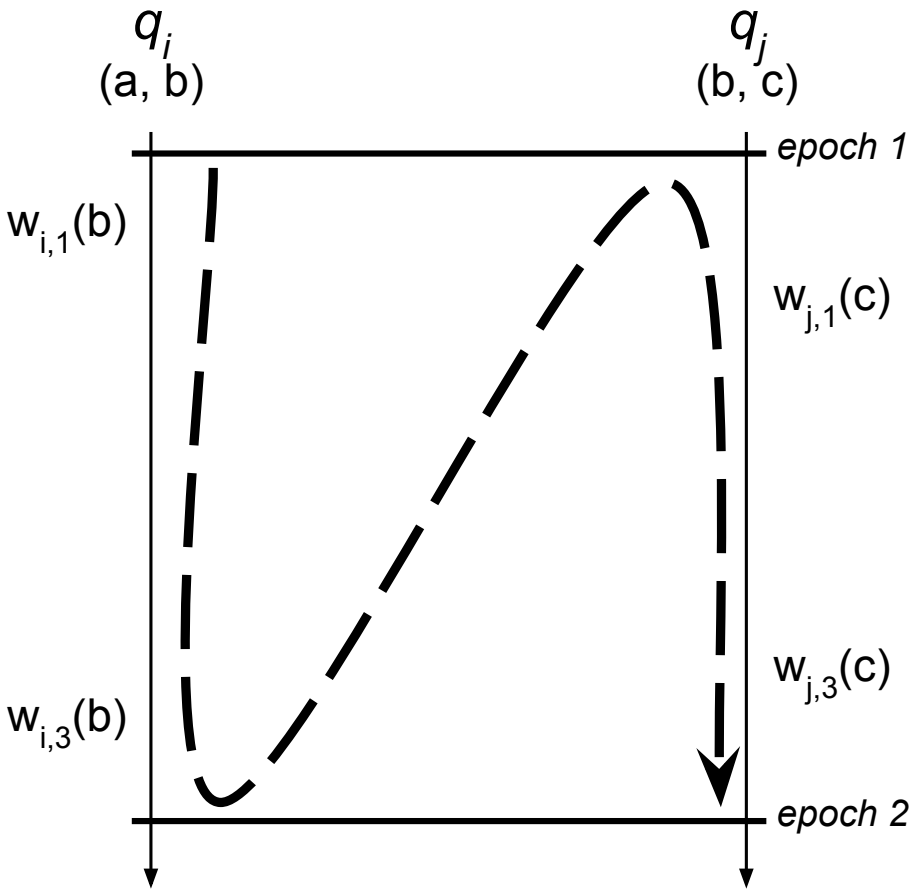


Figure 3.6: Without remote accesses, once an epoch has been concluded a default total ordering is: $w_{i,1} \rightarrow w_{i,3} \rightarrow w_{j,1} \rightarrow w_{j,3}$. Once the epoch is concluded, this ordering is guaranteed and exposes an externalizable total ordering of events.

grid – therefore at the epoch's conclusion a linearizable total ordering exists, though it cannot be read from. However, because only a single subquorum will handle reads within an epoch (no other quorum will issue a read unless the epoch has changed) we posit that a client that reads only objects handled by a single subquorum has a linearizable view of those objects. Because linearizability is composable across objects, the system is linearizable.

The constraint that clients access only a single subquorum per epoch is acceptable in the common case, but because epoch changes require root quorum decisions, we allow cross-quorum communication via *remote accesses*. Figure 3.7 shows additional dependencies created by issuing remote writes to other subquorums: $w_{i,2} \rightarrow w_{j,3}$ and $w_{j,2} \rightarrow w_{i,3}$. Each remote write establishes a partial ordering between events of the sender before the sending of the write, and writes by the receiver after the write is received. Similar dependencies result from remote reads.

These dependencies cause the epochs to be logically split (not shown in picture). The receipt of write $w_{i,2}$ in $q_j$ causes $q_{j,1}$ to be split into $q_{j,1.1}$ and $q_{j,1.2}$. Likewise, the receipt of write $w_{j,2}$ into $q_i$ causes $q_i$ to be split into $q_{i,1.1}$ and $q_{i,1.2}$. Any topological sort of the subepochs that respects these orderings, such as $q_{i,1.1} \rightarrow q_{j,1.1} \rightarrow q_{j,1.2} \rightarrow q_{i,1.2}$, results in a valid SC ordering.

As before, presenting a sequentially consistent global log across the entire system, then, only requires tracking these inter-subquorum data accesses, and then performing an $\mathcal{O}(n)$ merge of the subepochs. By definition, this log's ordering respects any externally visible ordering of cross-subquorum accesses (accesses visible to the system). So long as these dependencies are maintained, then the log is

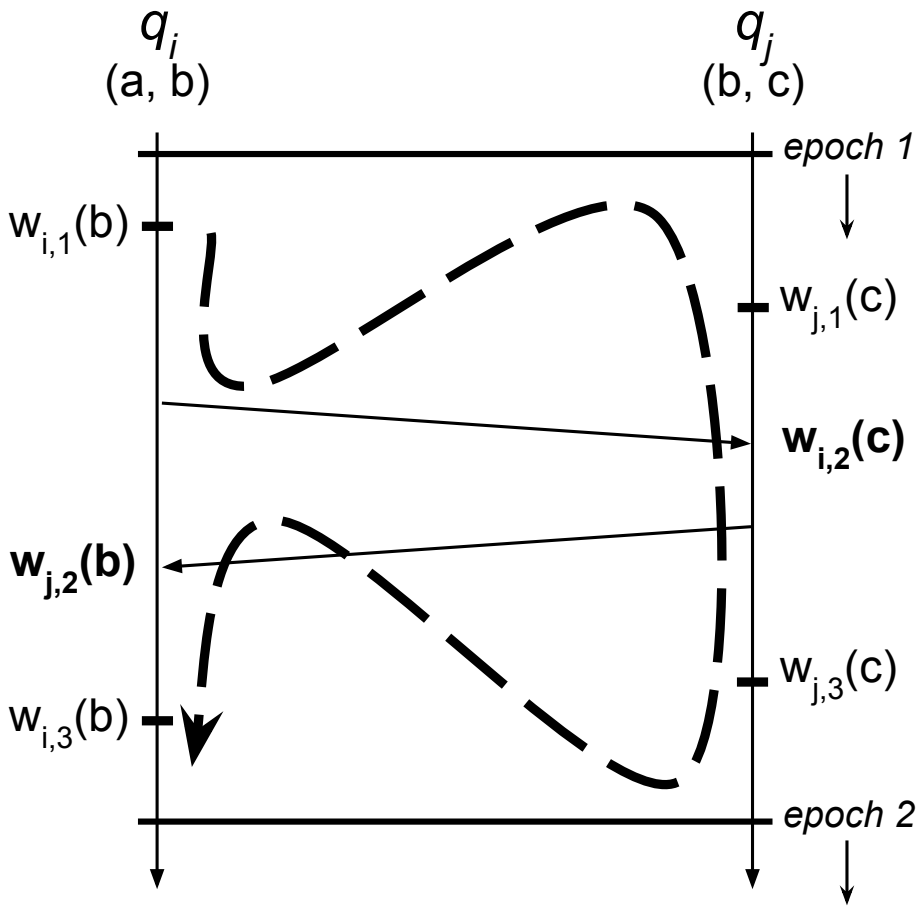


Figure 3.7: Remote writes add additional ordering constraints: $w_{i,1} \rightarrow w_{i,2} \rightarrow w_{j,3}$, and $w_{j,1} \rightarrow w_{j,2} \rightarrow w_{i,3}$. By creating *subepochs*, we can guarantee linearizability even for accesses across multiple subquorums.

externalized at epoch boundaries.

However, the log does not necessarily order other accesses according to external visibility. Extracting a global log could not be mined to find causal relationships between accesses through external communication paths unknown to the system. For example, assume that log events are published posts, and that one user claimed plagiarism. The accused would not be able to prove that his post came first unless there were some causal chain of posts and references visible to the protocol.

### 3.3.2 Globally Consistent Logs

Our default use case is in providing linearizable access to an object store. Though this approach allows us to guarantee all observers will see linearizable results of object accesses in real-time, the system is not able to enumerate a total order, or create a linearizable shared log. Such a linear order would require fine-grained (expensive) coordination across the entire system, or fine-grained clock synchronization [41]. Though many or most distributed applications (objects stores, file systems, etc.) will work directly with HC, shared logs are a useful building block for distributed systems.

HC *can* be used to build a sequentially consistent (SC) shared log as shown in Figure 3.8. Like Lin, SC requires all observers to see a single total ordering. SC differs in that this total ordering does not have to be externalizable. Instead, it merely has to conform to local operation orders and all reads-from dependencies created by remote accesses.

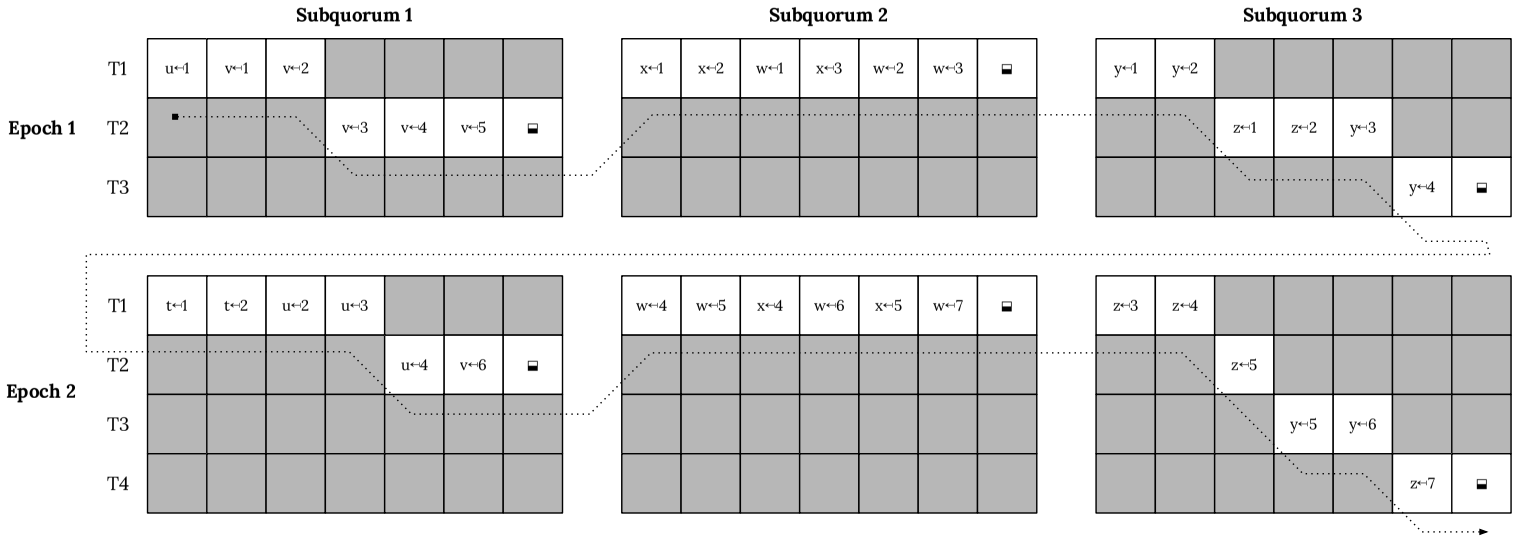


Figure 3.8: A sequential consistency view of a a global grid ordering. This figure shows the grid consistency model and happens before relationships between epoch configurations and writes between logs. This grid only exposes sequential consistency across all objects, however, because log operations are concurrent across all rows.

We do not currently gather the entire shared log onto a single replica because of capacity and flexibility issues. Capacity is limited because our system and applications are expected to be long-lived. Flexibility is a problem because HC, and applications built on HC, gain much of their value from the ability to pivot quickly, whether to deal with changes in the environment or for changing application access patterns. We require handoffs to be as lightweight as possible to preserve this advantage.

## 3.4 Safety and Correctness

Distributed consensus requires provable safety and correctness so as to be relied upon when building consistency-centric systems. Safety ensures that any update to the system, if committed, will be represented by the system even in the case of limited failure [121]. Correctness is described by the consistency model, if a distributed algorithm always produces an expected system state, then it is correct [122]. Safety and correctness are proved as part of the scientific process of introducing new consensus algorithms. Although we view hierarchical consensus as a consensus protocol rather than a new consensus algorithm, we provide a brief overview of our safety proof as follows.

We assert that consensus at individual subquorums is correct and safe because decisions are implemented using well-known leader-oriented consensus approaches. Hierarchical consensus therefore has to demonstrate linearizable correctness and safety between subquorums for a single epoch and between epochs. Briefly, lineariz-

ability requires external observers to view operations to objects as instantaneous events. Within an epoch, subquorum leaders serially order local accesses, thereby guaranteeing linearizability for all replicas in that quorum. Remote accesses and the internal subepoch invariant also enforce linearizability of accesses between subquorums inside of a single epoch as described in § 3.3.1. Given a static system of subquorum configurations that each manage independent shards, we claim that our system implements vertical paxos.

A static configuration would not require a root quorum, but it would also not allow reconfiguration to move quorums to locales of access or to repair system failures. Therefore to prove safety and correctness, we must show that root quorum behavior, specifically epoch transitions and delegation, is correct. Epoch transitions raise the possibility of portions of the namespace being re-assigned from one subquorum to another, with each subquorum making the transition independently. Correctness is guaranteed by an invariant requiring subquorums to delay serving newly acquired portions of the namespace until after completing all appropriate handshakes. Tombstones ensure that an update cannot be applied to a subquorum then lost when the transitioning subquorum takes over. Delegation is protected by bookkeeping that ensures that no replica can be counted twice in a vote, therefore in the worst case, delegation means that a single failure can eliminate many votes. We propose formal specifications in Appendix A to prove that these invariants are sufficient for correctness.

Safety and correctness are important parts of distributed consensus, but only if they also allow a system to make progress in the event of failure. We define the

system's *safety* property as guaranteeing that non-linearizable (or non-sequentially-consistent, see Section 3.3.2) event orderings can never be observed. We define the system's *progress* property as the system having enough live replicas to commit votes or operations in the root quorum. In the rest of this section, we will specifically identify types of expected failures that may harm our proposed guarantees and what amount of failure is tolerated before preventing progress. We then describe two additional mechanisms that we use to ensure the safety of hierarchical consensus: the *nuclear option* and *obligation leases*.

### 3.4.1 Fault Tolerance

The system can suffer several types of failures, as shown in Table 3.1. Both subquorum leaders and the root leaders send periodic beacons and heartbeat messages to their followers. If a heartbeat message is missed, e.g. if a follower does not receive an expected heartbeat from its leader or if a leader does not receive a response from the heartbeat, then the system takes action to ensure it's still available to respond to clients. Failures of subquorum and root quorum leaders are handled through the normal consensus mechanisms and a new leader is elected. Failures of subquorum peers are handled by the local leader petitioning the root quorum to re-configure the subquorum in the next epoch. Failure of a root quorum peer is the failure of a subquorum leader with delegated votes, this can be handled by a reconfiguration which reallocates the delegated votes to a new peer so long as a majority of delegates are available in the root quorum. Root quorum beacon messages help

inform replicas of leadership and configuration changes, which ensures the system adapts to temporary outages and failures.

Table 3.1: Failures include either node failure or network partitions which are detected by missed beacon or heartbeat messages. A replicas role and a threshold for the number of missed messages determines how the system responds.

| Failure Type | Response |
|---|---|
| subquorum peer | request reconfiguration from root quorum |
| subquorum leader | local election, request replacement from root quorum |
| subquorum | reconfiguration after obligations timeout |
| root leader | root election (with delegations) |
| majority of delegates | delegations time out (nuclear option) |

HC's structure means that some faults are more important than others. Proper operation of the root quorum requires the majority of replicas in the majority of subquorums to be non-faulty. Given a system with $2m+1$ subquorums, each of $2n+1$ replicas, the entire system's progress can be halted with as few as $(m+1)(n+1)$ well-chosen failures – e.g. the assassination of the delegates. Therefore, in worst case, the system can only tolerate: $f_{worst} = mn + m + n$ failures and still make progress. At maximum, HC's basic protocol can tolerate up to: $f_{best} = (m+1)*n+m*(2n+1) = 3mn + m + n$ failures. As an example a 25/5 system, that is a system of 25 replicas with size 5 subquorums ($m = 2, n = 2$), can tolerate at least 8 and up to 16 failures. A 21/3 system can tolerate at least 7, and a maximum of 12, failures out of 21 total replicas.

To achieve best possible fault tolerance, the root quorum operates strategically to handle failures. For example, individual subquorums might still be able to perform local operations despite an impasse at the global level. The root quorum

chooses carefully whether a failure type should involve a reconfiguration or whether the system should wait for an outage to be repaired. There are two primary types of failures though that have to be dealt with specifically. Total subquorum failure can temporarily cause a portion of the namespace to be unserved (or only served locally). In this case we use obligation timeouts to determine when the root quorum should force a configuration change. Additionally in the face of delegate assassination, where no root quorum decisions can be made, we use the nuclear option to eliminate delegates and require every replica to contribute their own vote.

### 3.4.2 The Nuclear Option

Singleton consensus protocols, including Raft, can tolerate just under half of the entire system failing. As described above, HC's structure makes it more vulnerable to clustered failures. Therefore we define a *nuclear option*, which uses direct consensus decision making among all system replicas to tolerate any $f$ replicas failing out of $2f + 1$ total replicas in the system.

A nuclear vote is triggered by the failure of a root leader election. A *nuclear candidate* increment's its term for the root quorum and broadcasts a request for votes to all system replicas. The key difficulty is in preventing delegated votes and nuclear votes from reaching conflicting decisions. Such situations might occur when temporarily unavailable subquorum leaders regain connectivity and allow a wedged root quorum to unblock. Meanwhile, a nuclear vote might be concurrently underway.

Replica delegations are defined as intervals over specific slots. Using local subquorum slots would fall prey to the above problem, so we define delegations as a small number (often one) of root slots, which usually correspond to distinct epochs. During failure-free operation, peers delegate to their leaders and are all represented in the next root election or commit. Peers then renew their delegations to their leaders by appending them to the next local commit reply. This approach works for replicas that change subquorums over an epoch boundary, and even allows peers to delegate their votes to arbitrary other peers in the system (see replicas $r_N$ and $r_O$ in Figure 3.3).

This approach is simple and correct, but deals poorly with leader turnovers in the subquorum. Consider a subquorum where all peers have delegated votes to their leader for the next root slot. If that leader fails, none of the peers will be represented. We finesse this issue by re-defining such delegations to count root elections, root commits, *and* root heartbeats. The latter means that local peers will regain their votes for the next root quorum action if it happens after the next heartbeat.

Consider the worst-case failure situation discussed in § 3.4.1: a majority of the majority of subquorums have failed. None of the failed subquorum leaders can be replaced, as none of those subquorums have enough local peers.

The first response is initiated when a replica holding delegations (or its own vote) times out waiting for the root heartbeat. That replica increments its own root term, adopts the prior system configuration as its own, and becomes a root candidate. This candidacy fails, as a majority of subquorum leaders, with all of

their delegated votes, are gone. Progress in the root quorum is not made until delegations time out. In our default case where a delegation is for a single root event, this happens after the first root election failure.

At the next timeout, any replica might become a candidate because delegations have lapsed (under our default assumptions above). Such a nuclear candidate increments its root term and sends candidate requests to all system replicas, succeeding if it gathers a majority across all live replicas.

The first candidacy assumed the prior system configuration in its candidacy announcement. This configuration is no longer appropriate unless some of the "failed" replicas quickly regain connectivity. Before the replica announces its candidacy for a second time, however, many of the replica replies have timed out. The candidate alters its second proposed configuration by recasting all such replicas as hot spares and potentially reducing the number and size of the subgroups. Subsequent epoch changes might re-integrate the new hot spares if the replicas regain connectivity.

### 3.4.3 Obligations Timeout

The second type of specific failure the root quorum must deal with is a subquorum stranded behind a network partition. In this case the subquorum may be operating and serving local requests but its leader (or delegate) is unable to communicate to the root quorum. In the majority of cases, the root quorum should wait out the presumably temporary system partition if client requests are being served. However, it is also possible that all replicas in the subquorum have failed due to

cascading correlated failure and no accesses to that portion of the namespace are being granted.

We therefore propose a configurable obligations timeout. Subquorums are considered obligated to serve requests from clients for the duration of the epoch in which the subquorum is configured. However to ensure that subquorums are in fact meeting those obligations, we introduce another timeout during which the subquorum has to communicate with the root leader, reconfirming its obligation for the next period. If the obligation period times out without being renewed, the subquorum is obligated to stop handling client requests and the root quorum is obligated to reconfigure the system.

The problem is that the subquorum that is reallocated that portion of the namespace presumably would not be able to achieve a hand-off with the partitioned system. It is also possible that the region the subquorum was configured in simply cannot be reconfigured through a root consensus decision. In this case, there would be an unacceptable period of unavailability for that portion of the namespace. To deal with this situation, both the newly configured subquorum and the previous subquorum must go into an "unstable" state, informing clients that their writes are not guaranteed the level of consistency the system normally provides. Using a federated consistency model, replicas would simply assume a lower level of consistency. Unstable mode is repaired in one of two cases, either the partitioned subquorum comes back on line and is able to automatically negotiate the epoch transition, fixing conflicts where necessary or it manually determined that the subquorum had been destroyed, which results in data loss anyway. Optimizations such

as anti-entropy replication of data across regions and global views of data versions minimize the impact of such loss, but cannot prevent it.

## 3.5 Performance Evaluation

HC was designed to adapt both to dynamic workloads as well as variable network conditions. We therefore evaluate HC in two distinct environments: a homogeneous data center and a heterogeneous real-world network. The homogeneous cluster is hosted on Amazon EC2 and includes 26 "t2.medium" instances: dual-core virtual machines running in a single VPC with inter-machine latencies of $\lambda_\mu = 0.399ms$ and $\lambda_\sigma = 0.216ms$. These machines are cost effective and, though lightweight, are easy to scale to large cluster sizes as workload increases. Experiments are set up such that each instance runs a single replica process and multiple client processes.

The heterogeneous cluster (UMD) consists of several local machines distributed across a wide area, with inter-machine latencies ranging from $\lambda_\mu = 2.527ms$, $\lambda_\sigma = 1.147ms$ to $\lambda_\mu = 34.651ms$, $\lambda_\sigma = 37.915ms$. Machines in this network are a variety of dual and quad core desktop servers that are solely dedicated to running these benchmarks. Experiments on these machines are set up so that each instance runs multiple replica and client processes co-located on the same host. In this environment, localization is critical both for performance but also to ensure that the protocol can elect and maintain consensus leadership. The variability of this network also poses challenges that HC is uniquely suited to handle via root quorum-guided

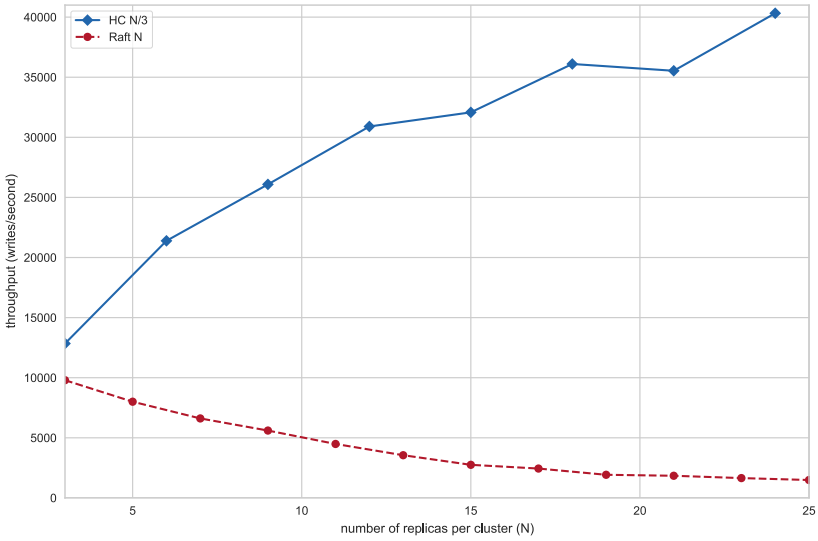


Figure 3.9: Mean throughput of workloads of up to 120 concurrent clients

adaptation. We explore two distinct scenarios – sawtooth and repartitioning – using this cluster; all other experiments were run on the EC2 cluster.

HC is partially motivated by the need to scale strong consistency to large cluster sizes. We based our work on the assumption that consensus performance decreases as the quorum size increases, which we confirm empirically in Figure 3.9. This figure shows the maximum throughput against system size for a variety of workloads, up to 120 concurrent clients. A workload consists of one or more clients continuously sending writes of a specific object or objects to the cluster without pause.

Standard consensus algorithms, Raft in particular, scale poorly with uniformly decreasing throughput as nodes are added to the cluster. Commit latency increases

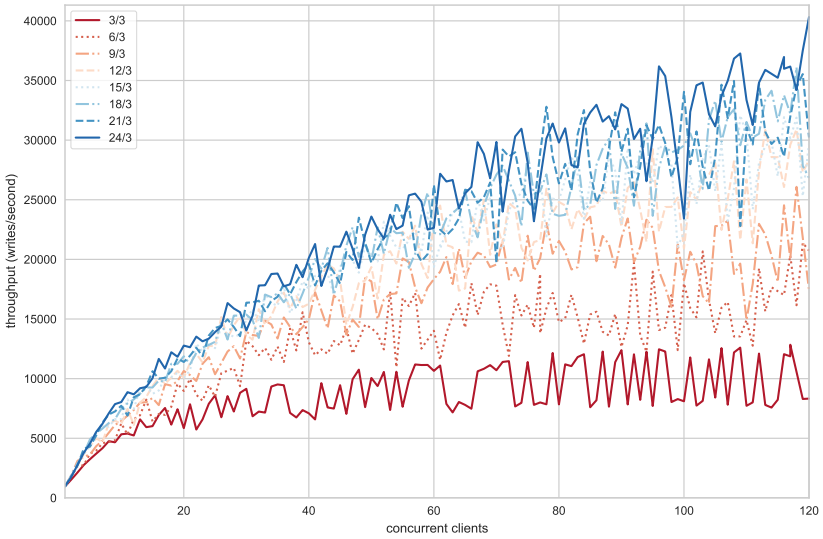


Figure 3.10: Performance of distributed consensus with an increasing workload of concurrent clients. Performance is measured by throughput, the number of writes committed per second.

with quorum size as the system has to wait for more responses from peers, thereby decreasing overall throughput. Figures 3.9 and 3.10 clearly show the multiplicative advantage of HC's hierarchical structure, though HC does not scale linearly as we had expected.

There are at least two factors currently limiting the HC throughput shown here. First, the HC subquorums for the larger system sizes are not saturated. A single 3-node subquorum saturates at around 25 clients and this experiment has only about 15 clients per subquorum for the largest cluster size. We ran experiments with 600 clients, saturating all subquorums even in the 24-node case. This throughput peaked at slightly over 50,000 committed writes per second, better but still lower

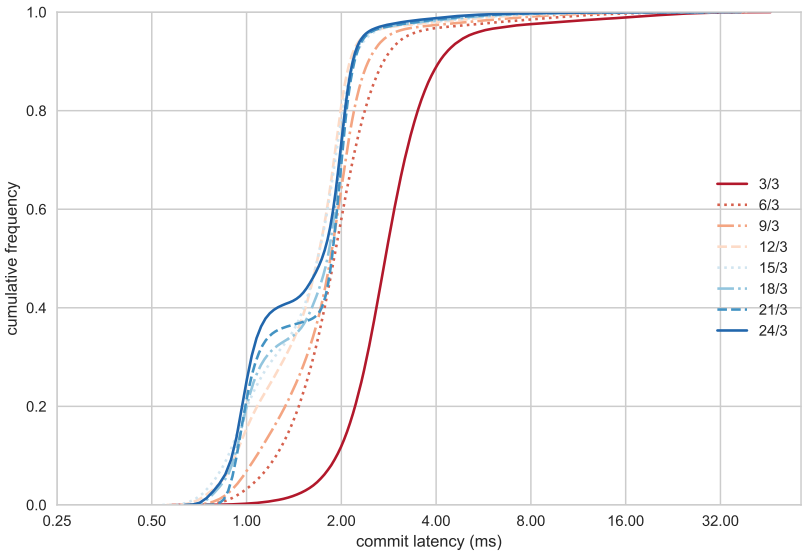


Figure 3.11:

than the linear scaling we had expected.

We think the reason for this ceiling is hinted at by Figure 3.10. This figure shows increasingly larger variability with increasing system sizes. A more thorough examination of the data shows widely varying performance across individual subquorums in the larger configurations. We suspect that the cause is either VM misconfiguration or misbehavior. We are adding more instrumentation to diagnose the problem.

The effect of saturation is also demonstrated in Figure 3.11, which shows cumulative latency distributions for different system sizes holding the workload (number of concurrent clients) constant. The fastest (24/3) shows nearly 80% of client write requests being serviced in under 2 msec. Larger system sizes are faster because the

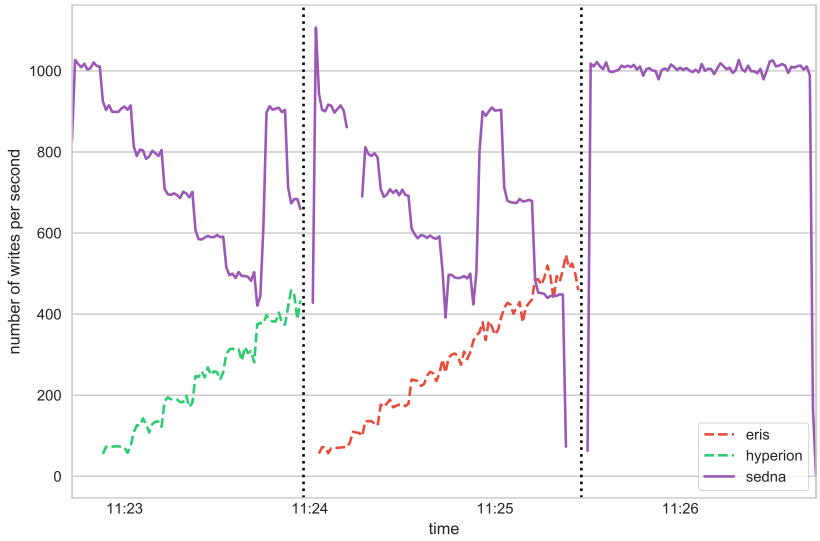


Figure 3.12: 9/3 system adapting to changing client access patterns by repartitioning the tag space so that clients are co-located with subquorums that serve tags they need.

smaller systems suffer from contention (25 clients can saturate a single subquorum). Because throughput is directly related to commit latency, throughput variability can be mitigated by adding additional subquorums to balance load.

Besides pure performance and scaling, HC is also motivated by the need to adapt to varying environmental conditions. In the next set of experiments, we explore two common runtime scenarios that motivate adaptation: shifting client workloads and failures. We show that HC is able to adapt and recover with little loss in performance. These scenarios are shown in Figures 3.12 and 3.13 as throughput over time, where vertical dotted lines indicate an epoch change.

The first scenario, described by the time series in Figure 3.12 shows an HC

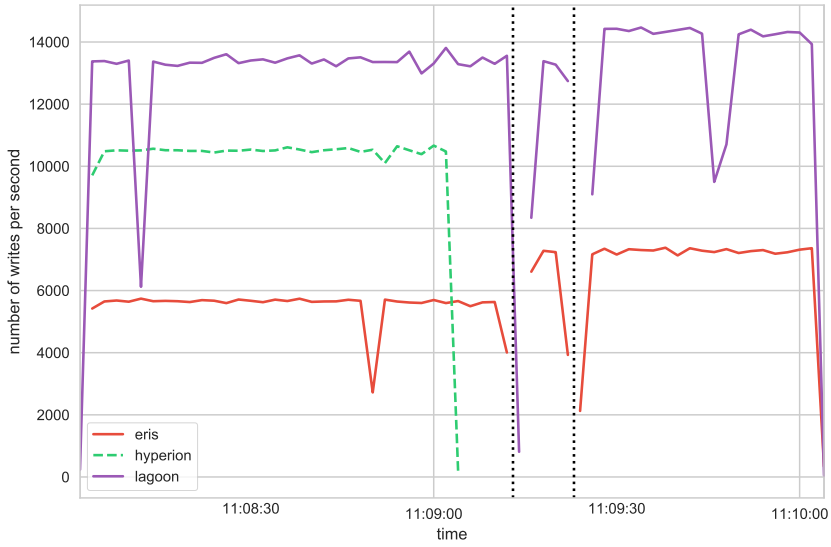


Figure 3.13: 9/3 System that adapts to failure (partition) of entire subquorum. After timeout, the root quorum re-partitions the tag allocated to the failed subquorum among the other two subquorums.

3-replica configuration moving through two epoch changes. Each epoch change is triggered by the need to localize tags accessed by clients to nearby subquorums. The scenario shown starts with all clients co-located with the subquorum serving the tag they are accessing. However, clients incrementally change their access patterns first to a tag located on one remote subquorum, and then to the tag owned by the other. In both cases, the root quorum adapts the system by repartitioning the tagspace such that the tag defining their current focus is served by the co-located subquorum.

Finally, Figure 3.13 shows a 3-subquorum configuration where one entire subquorum becomes partitioned from the others. After a timeout, the root uses an epoch change to re-allocate the tag of the partitioned subquorum over the two

remaining subquorums. The partitioned subquorum eventually has an *obligation timeout*, after which the root quorum is not obliged to leave the tag with the current subquorum. The tag may then be re-assigned to any other subquorum. Timeouts are structured such that by the time an obligation timeout fires, the root quorum has already re-mapped that subquorum's tag to other subquorums. As a result, the system is able to recover from the partition as fast as possible. In this figure, the repartition occurs through two epoch changes, the first allocating part of the tagspace to the first subquorum, and the second allocating the rest of the tag to the other. Gaps in the graph are periods where the subquorums are electing local leaders. This may be optimized by having leadership assigned or maintained through root consensus.

## 3.6 Conclusion

Most consensus algorithms have their roots in the Paxos algorithm, originally described in parliamentary terms. The metaphor of government still applies well as we look at the evolution of distributed coordination as systems have grown to include large numbers of processes and geographies. Systems that use a dedicated leader are easy to reason about and implement, however, like chess, if the leader goes down the system cannot make any progress. Simple democracies for small groups solve this problem but do not scale, and as the system grows, it fragments into tribes. Inspired by modern governments, we have proposed a representative system of consensus, hierarchical consensus, such that replicas elect leaders to participate

in a root quorum that makes decisions about the global state of the system. Local decision making, the kind that effects only a subset of clients and objects is handled locally by subquorums as efficiently as possible. The result is a a hierarchy of decision making that takes advantage of hierarchies that already exist in applications.

Hierarchical Consensus is an implementation and extension of Vertical Paxos. Like Vertical Paxos, HC reasons about consistency across all objects by identifying commands with a grid ordering (rather than a log ordering) and is reconfigurable to adapt to dynamic environments that exist in geo-replicated systems. Adaptability allows HC to exploit locality of access, allowing for high performance coordination, even with replication across the wide area. HC extends Vertical Paxos to ensure that intersections exist between the subquorums and the root quorum, to guarantee operations between subquorums, and to ensure that the system operates as a coordinated whole. To scale the consensus protocol of the root quorum, we propose a novel approach, delegation, to ensure that all replicas participate in consensus but limit the number and frequency of messages required to achieve majority. Finally, we generalized HC from primary-backup replication to describe more general online replication required by distributed databases and file systems.

In the next chapter we will explore a hybrid consistency model implemented by federating replicas that participate in different consistency protocols. In a planetary scale network, HC provides the strong consistency backbone of the federated model, increasing the overall consistency of the system by making coordinating decisions at a high level, and allowing high availability replicas in the fog operate independently where necessary.

# Chapter 4: Federated Consistency

The next generation of globally distributed systems will not only reside in carefully managed cloud data centers connected by multiple communication trunks. To handle increasing mobile demand, application services have migrated closer to the edge of the computing environment [123]. Non-human users and machine-to-machine communication will also require geography-specific data systems to handle traffic coordination and electrical grid data [6–10]. To support these trends, a fog of partial-replicas that serve clients in specific extra-datacenter locations is required to bridge the gap between the centralizing tendency of the cloud and the decentralizing tendency of edge computing [124–126]. For that reason, we propose that in addition to first tier strong-consistency backbone, a planetary-scale distributed system also requires a second-tier dissemination network that provides a high-availability mesh between quorum decision making [1].

Outside of a data center context, strong coordination using consensus is simply not feasible [127]. In a stable network environment, systems are able to adapt consistency at runtime [128–132]. Systems outside of this context are expected to have heterogenous hardware which leads to variability both in terms of capacity and failure rates. The farther the network diameter, the more partition prone a

network becomes, and the higher latencies are experienced between nodes. Mobility also means dynamic membership with many peers, therefore even adaptable configuration provided by hierarchical consensus is not sufficient. Taken together, these challenges require a minimization of coordination, instead a focus on cacheing and a high-throughput of writes to the rest of the system. In other words, a relaxation of consistency guarantees in the fog layer.

In this chapter, we present a novel approach to flexible consistency that federates replicas that participate in a system with different consistency guarantees. By allowing individual replicas to maintain strong consistency for their clients if they are part of a strongly connected part of the network (e.g. in a datacenter or in the cloud) or to relax consistency guarantees if they are in a more variable network, we ensure that the entire system behaves as a single, integrated entity. Individual replicas in the system are allowed to adapt to a changing network environment while providing as strong a local guarantee or minimum quality of service as required. The global state of a federated system is defined by the replica topology and their interactions. If a subset of replicas implement strong consistency models such has hierarchical consensus, then the global probability of conflict is reduced. Conversely, a subset of replicas implementing weaker consistency can increase global throughput. We find that it is more often the tension between local vs. global views of consistency that cause the greatest concerns about application performance. Because each node can select and change local consistency policies, client applications local to the replica server have greater control of tuning consistency, maximizing timeliness or correctness as needed.

A federated consistency protocol can find a middle ground in the trade-off between performance and consistency. In this chapter we consider two extremes: an eventually consistent system implemented with gossip-based anti-entropy [37,78] and a sequential consistency model as implemented by the Raft consensus prtocol [92] (we use Raft as a stand-in for hierarchical consensus to simplify the discussion in this chapter). By exploring these two extremes in the consistency spectrum we show that the overall number of inconsistencies in the system is reduced over a homogenous eventual system, and that the access latency is decreased from the homogenous sequential system.

## 4.1 Hybrid Consistency

Federating replicas that participate in multiple protocols leading to different consistency levels creates a *hybrid consistency* model [133]. Hybrid consistency models attempt to use strong consistency with application semantics demand it and weak consistency when not required. We propose a hybridization not due to required semantics, but rather based on network environment. Our consistency model is therefore topology-dependent and more than simply hybridizing consistency, can be said to have flexible or dynamic consistency. We have found that large systems with variable latency in different geographic regions can perform well by allowing most nodes to operate optimistically, but also maintaining a strong central quorum to reduce the amount of global conflict.

In § 2.2.1 we defined a data-centric consistency model that viewed consistency

in terms of per-replica logs that describe the sequence of operations that modify a replica's state. Consistency models express the correctness of a system based on two metrics: the strictness of log ordering and how stale a log is allowed to be [74]. *Ordering* refers to how closely individual logs adhere to an abstract global ordering. Strong consistency requires all logs to be identically ordered, and consensus algorithms coordinate a majority of replicas to correctly appended entries to the log in the same order. Weak consistency allows divergence of the order operations applied to the log.

On the other hand, *Staleness* refers to how far local logs are behind the latest version of the global log, which can be expressed by the average latency of replicating updates, or how far behind the average replica is from the log. Most data-centric models do not consider staleness, instead referring to guarantees on ordering strictness. However, the symptoms of inconsistency are primarily due to staleness [93, 94].

Consider a system where each update creates a new version of the object that maintains the parent version the update was created from. In a distributed system with multiple nodes, two forms of inconsistencies can occur. First, a *stale read* occurs if the version read is not the latest global version, e.g. it is incorrect that reading from two parts of the system may return different answers. Second, stale reads lead to *forks* on updates: when two replicas concurrently write a new version to the same parent object. Forks introduce inconsistency because they allow multiple potential orderings of operation logs and from the client's perspective, because they introduce conflicts.

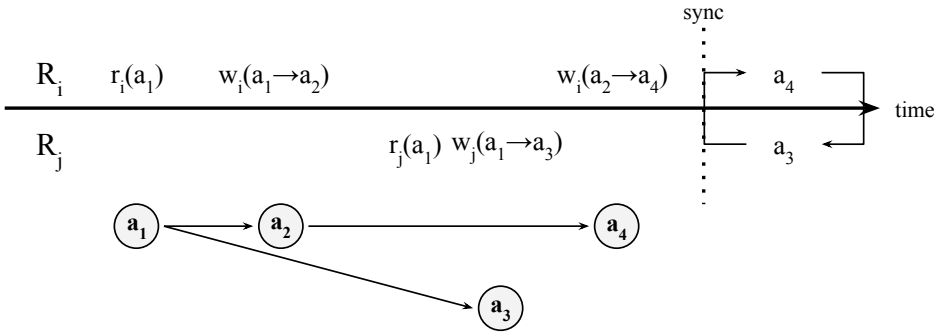


Figure 4.1: Accesses before synchronization cause stale reads and forked writes. In this case if $R_i$ and $R_j$ both attempt to write object a at version 1, $a_1$, the result will be two new versions, $a_2$ and $a_3$ both of which have the parent version $a_1$, which could mean a potential conflict.

Object *coherence* requires an objects' version history to be a linear sequence [134], which demonstrates that the system was in a consistent state during all accesses. Version history forks violate coherence and occur, for example, when replicas $i$ and $j$ read object version $a_1$ then concurrently attempt to write new versions: $W_i(a_1 \rightarrow a_2)$ and $W_j(a_q \rightarrow a_3)$ as shown in Figure 4.1. A delay in synchronization between $R_i$ and $R_j$ could lead one replica to continue farther down the fork paths, e.g. $W_i(a_2 \rightarrow a_4)$. Forks can be caused by concurrent reads, but the fork between $a_2$ and $a_3$ actually occurs because $R_j$'s read is stale. Forks are a useful model for exposing how ordering and staleness are considered in different consistency approaches.

In an *eventually consistent* system, a replica's log does not depend on any other replica's log except that the last entry appended must eventually be identical for each object in the namespace. In practice, this means that eventually consistent logs keep track of monotonically increasing versions and that not all versions are

required to be present in the log, so long as the same final state is achieved. This suggests that eventual consistency requires a synchronization mechanism to propagate writes asynchronously and a policy to handle convergence [117]. Forks occur in the eventually consistent model because it optimistically accepts writes without very much coordination, allowing to concurrent versions to be appended to two different logs. Generally speaking, conflict resolution is left to the application layer, but in practice each conflict must be resolved as it reaches each replica.

Eventually consistency convergence is typically implemented by a *last writer wins policy*. When replicas synchronize, they compare the latest version of each object based on all updates prior to their synchronization, then accept whichever version is latest. As a result, eventually consistent logs may temporarily diverge, so long as the final version of objects eventually converge. It is therefore acceptable to alternate between writes to competing forks (a fairly weak semantic) or to drop a branch with more updates in favor of a more recent, shorter branch. In § 4.2.1 we will describe in more detail the likelihood of forks in bilateral anti-entropy.

In a *sequentially consistent* system, the ordering of updates to individual objects must be identical and no versions must be missing. However, it is possible that the logs of lagging replicas may only be prefixes of the latest log in the system [135]. Sequential consistency therefore does not make guarantees about staleness (or the ordering of reads) but does require all writes become visible in the same order [74]. Sequential consistency can be implemented with consensus algorithms such as Paxos [60] or Raft [92], which coordinate logs by defining a transitive global ordering for all conflicts. Alternatively, sequential consistency can be implemented

with warranties – time-based assertions about groups of objects that must be met on all replicas before the assertions expire [136]. In both implementations, forks can be immediately observed because sequential consistency requires coordination when any update occurs.

Stale reads in a sequentially consistent system are possible because of lagging replicas. However, only a single branch of a forked write can be committed to any copy of the log. Preventing forks would require either a locking mechanism or an optimistic approach that allowed operations to occur but rejects all but one branch. Therefore in a sequentially consistent system implemented by consensus, forks are rejected and appear to the user as dropped writes, at which point the client must retry or immediately resolve conflicts. In § 4.2.2 we investigate cache read policies for consensus that modulate the probability of a dropped write.

At both ends of the consistency spectrum we have presented, eventual and sequential consistency, it is clear that flexibility is primarily in the *amount* of forks that may occur. In a hybrid model, we therefore express consistency as a likelihood that a fork occurs. If a client accesses a strong consistency replica, the likelihood is low and the client is immediately notified of a conflict. If the access is to an eventually consistent replica, the likelihood is higher, and conflicts must be handled after the access. Our key insight is that eventually consistent replicas participating in a federated system do not affect the likelihood or behavior of consistency at sequentially consistent replicas. Although we have discussed strong and weak consistency models, this observation also applies to other consistency models such as causal consistency [79,80,137,138], for simplicity however, we continue our discussion with

only strong and weak models.

## 4.2 Replication

A federated consistency model allows individual replicas to engage in replication according to locally-specified consistency policies. Each replica maintains its own local state, modified in response to local accesses and receipt of messages from remote replicas. Each replica sends messages to other replicas to propagate new writes. Every federated replica must have the ability to handle all types of RPC messages required by different protocols and each protocol must be expressed by separate RPC endpoints. So long as this is true, then federation primarily has to be defined at the *consistency boundaries*, that is when replicas of one consistency type send messages to that of another.

We consider a system where clients can `Put` values (write) and `Get` (read) independent objects specified by a key. `Get` requests are fulfilled by reading from the local cache of a replica depending on its read policy. On `Put`, a new instance of the object is created and assigned a monotonically increasing, conflict-free *version number* [139, 140]. For simplicity, we assume a fixed number of replicas, therefore each version is made up of two components: the *update* and *precedence ids*. Precedence ids are assigned to replicas during configuration, and update ids are incremented to the largest observed value during synchronization. As a result, any two versions generated by a `Put` anywhere in the system are comparable such that the *latest* version of the key-value pair is the version with the largest update id, and in the

case of ties, the largest precedence id.

For simplicity, we assume that all object instances must become fully replicated to the entire system. In practice we observe that metadata about the objects usually becomes fully replicated, which points to the locations the object data is stored for retrieval. Consistency models define how replication occurs. An eventually consistent model propagates updates asynchronously using gossip-based anti-entropy to synchronize pairs of replicas without congesting the network with broadcasts. Consensus, on the other hand, replicates the object and commits it before the access is completed.

### 4.2.1 Gossip-Based Anti-Entropy

Eventual consistency is implemented using read/write quorums and background anti-entropy. In this model, clients select one or more replicas to perform a single operation. The set of replicas that responds to a client creates a quorum that must agree on the state of the operation at its conclusion. Clients can vary read and write quorum sizes to improve consistency or availability – larger quorums reduce the likelihood of inconsistencies caused by concurrent updates, but smaller quorums respond much more quickly, particularly if the replicas in the quorum are co-located with the client. In large, geo-replicated systems we assume that clients will prefer to choose fewer, local replicas to connect with, optimistic that collisions across the wide-area are rare, e.g. that writes are localized but reads are global. We therefore primarily focus on anti-entropy synchronization.

As clients make accesses to individual replicas, their state diverges as they follow independent object version histories. If allowed to remain wholly independent, individual requests from clients to different replicas would create a lack of order or predictability, a gradual decline into inconsistency, e.g. the system would experience *entropy*. To combat the effect of entropy while still remaining highly available, servers engage in periodic background anti-entropy sessions [37, 78, 117]. Anti-entropy sessions synchronize the logs of two replicas ensuring that, at least briefly, the local state is consistent with a portion of the global state of the system. If all servers engage in anti-entropy sessions, the system will converge barring any accesses that produce entropy.

Anti-entropy is conducted using gossip protocols such that pairs of replicas synchronize each other on a periodic interval to ensure that the network is not saturated with synchronization requests that may reduce client availability [141–143]. At each interval, every replica selects a synchronization partner such that all replicas have a uniform likelihood of selection. This ensures that an update originating at one replica will be propagated to all online replicas given the continued operation of replication. This mechanism also provides robustness in the face of failure; a single unresponsive replica or even network partition does not become a bottleneck to synchronization, and once the failure is repaired synchronization will occur without reconfiguration.

There are two basic forms of synchronization: *push* synchronization is a fire-and-forget form of synchronization where the remote replica is sent the latest version of all objects, whereas *pull* synchronization requests the latest version of objects and

minimizes the size of data transfer. To get the benefit of both, we consider *bilateral* synchronization which combines push and pull in a two-phase exchange. Bilateral synchronization increases the effect of anti-entropy during each exchange because it ensures that in the common case each replica is synchronized with two other replicas instead of one during every anti-entropy period.

Bilateral anti-entropy starts with the initiating replica sending a vector of the latest local versions of all keys currently stored, usually optimized with Merkel tree [144] or prefix trie [145] to make comparisons faster. The remote replica compares the versions sent by the initiating replica with its current state and responds with any objects whose version is *later* than the initiating replica's as well as another version vector of requested objects that are earlier on the remote. The initiating replica then replies with the remote's requested objects, completing the synchronization. We refer to the first stage of requesting later objects from the remote as the pull phase, and the second stage of responding to the remote the push phase.

There are two important things to note about this form of anti-entropy exchange. First, this type of synchronization implements a *latest writer wins* policy. This means that not all versions are guaranteed to become fully replicated – if a later version is written during propagation of an earlier version, then the earlier version gets *stomped* by the later version because only the latest versions of objects are exchanged. If there are two concurrent writes, only one write will become fully replicated, the write on the replica with the greater precedence.

Forks are caused by staleness due to propagation delays. The *visibility latency* of anti-entropy synchronization can be modeled as:

$$t_{visibility} \approx \frac{T}{4} \log_3 N + \epsilon \tag{4.1}$$

The parameter $\frac{T}{4}$ represents the delay between anti-entropy sessions, e.g. the periodicity of synchronizations. This delay is parameterized by the stability of the network environment, informed by a *tick* parameter, $T$, which is discussed in § 4.2.3. Bilateral anti-entropy in the best case would exponentially propagate updates across the network, therefore the visibility latency would depend on how often synchronizations occur and the diameter of the network, expressed by the number of replicas, $N$. However, the randomness of peer selection, which ensures safety, also means that two replicas that do not require synchronization may select each other, causing additional latency and noise represented by $\epsilon$. If $\epsilon = 0$, this would mean that each replica perfectly selected another replica which had not seen the write being propagated. In Chapter 6, we describe a reinforcement learning approach to optimizing $t_{visibility}$.

### 4.2.2 Sequential Consensus

In this chapter we consider a sequential consistency model implemented by replicating the operation log through the Raft consensus algorithm [92]. In § 3.2, we briefly described Raft consensus, though we focused on its use to implement linearizablity rather than sequential consistency, in this section, we will briefly summarize the differences of consensus for relaxed consistency. Raft is a leader-oriented

consensus protocol that uses timing parameters to detect failures and ensure that a leader is available to handle requests with minimal downtime. The leader has the primary responsibility of serializing and committing new operations to the replicated log. To that end, the leader will broadcast periodic heartbeats to maintain its leadership for a given term.

All write accesses, even those that originate at followers, must be forwarded to the leader who arbitrates the order in which commands are appended by the order in which they are received. In this way, the leader can guarantee a sequential ordering of updates so long as a majority of followers agree to commit the entries in the log at the specified positions. Our implementation differs from generic implementations in that the leader is also responsible for detecting forks – a write having a parent version that is already listed as a parent version in the log. Because the leader arbitrates all writes, it has the ability to detect forks and can reject (drop) the later write.

Dropped writes suggest that clients must submit writes containing its version history, and that stale reads are possible. In Chapter 3 we described a linearizable mode of consensus where both reads and writes must be totally ordered through the leader. Sequential consistency relaxes this requirement, allowing reads to be responded to by the caches of the followers, introducing a potential delay between when an updated is committed at the leader and when the follower is notified of the commit. We therefore define several possible read polices:

1. `read_committed` – Raft replicas only read the latest committed version of an

object, which occurs at best on the *second* round of communication. Committed writes are guaranteed not to be rolled back, but introduces the most significant delay, increasing the likelihood of a fork in high throughput periods.

2. `read_latest` – Replicas read the latest version of the object in their log, even if it has yet to be committed. Additionally, replicas will read writes originating locally rather than waiting for the first round of leader communication. In this case, reads are fast, but may return values that are never committed.

3. `read_remote` – All reads become synchronous requests to the leader, which can either guarantee linearizablity or use either of the above cache policies. This introduces communication latency, but may be faster if the expected message latency is less than commit latency.

Each of these options has critical implications for the likelihood of stale reads and dropped writes. Replicas would choose `read_committed` if the network was highly partition prone and messages from the leader were unstable, causing leadership changes that would rollback updates. The `read_remote` policy serves quorums well when the average message latency is below the commit latency, which is why we chose it to implement the strongest possible consistency in our hierarchical consensus cloud-tier. At the edge, intuition suggested and experimentation confirmed that the `read_latest` is the most appropriate approach for sequential consistency when quorum leadership is expected to be stable.

### 4.2.3 Timing Parameters

Both the anti-entropy and Raft protocols are parameterized by timing constraints that govern replication. We posit that consistency depends on the environment and even though Raft safety doesn't depend on timing parameters, they do define its progress properties. We expect that in the fog environment, network conditions will be highly variable, therefore we propose that all time-related parameters are based on a "tick" parameter, $T$. The tick parameter is a function of the observed message latency in the system, specified as a normal distribution of latency described by its mean, $\lambda_\mu$ and standard deviation, $\lambda_\sigma$. As the latency distribution changes over windows of time, the tick parameter can be updated to optimize the system. $T$ therefore must be used to define all timing parameters, we use a conservative formulation that is big enough to withstand most variability:

$$T = 6(\lambda_\mu + 4\lambda_\sigma) \tag{4.2}$$

Most implementations of Raft use a more conservative tick parameter of $10\lambda_\mu$, causing replication to occur more slowly than access events and also causing large conflicts and outage periods [59, 92]. Other formulations are more optimistic in data centers with stable network connections, for example $2\,(\lambda_\mu + 2\lambda_\sigma)$ [146]. This formulation is intended to maximize availability on leader failure, but is too small to capture the variability of our target environment, leading to out-of-order messages, which rapidly degrade performance.

In a federated system, all timing parameters are defined in terms of $T$. For example, to ensure that eventual and sequential replicas send approximately the same number of messages, e.g. to fix the message budget in capacity constrained environments, timing parameters may be selected as follows: The Raft election timeout is set to $U(T, 2T)$, with a heartbeat interval of $\frac{T}{2}$ while the anti-entropy delay is $\frac{T}{4}$. In Chapter 5, we will also condition the timing parameters of hierarchical consensus on the tick.

## 4.3 Federation

A federated model of consistency creates heterogeneous clouds of replicas that participate in different replication protocols. Global consistency and availability of the system is tuned by specifying different allocations of replicas of each type. Allocating all of one replication protocol, e.g. a homogeneous eventual or consensus topology, should behave equivalently to a homogeneous system that does not implement federation. Therefore a key requirement of federated consistency is the integration of protocols with no performance cost to replicas participating at different, local consistency levels.

We expect that a federated model will allow an eventual fog layer to benefit from lower fork frequency by being connected to a strongly consistent, central consensus group. Similarly, strong consistency replicas should be able to use anti-entropy mechanisms to replicate data and continue writing even if the leader is unavailable and no consensus can be reached to elect a leader (this is one possible

solution to the obligations timeout problem described in § 3.4.3). We integrate each systems by relying on the eventual replicas to disseminate orderings and cope with failures, but relying on the consensus replicas to choose the final operation ordering. To achieve this with no performance cost we must ensure that replicas can inter-operate both in terms of communication (message traffic) and consistency.

### 4.3.1 Communication Integration

All replication protocols are defined by their RPC messages and expected responses. On one level it is a simple matter to integrate the communication across protocols by ensuring that all replicas respond to all RPC message types, and that those types are clearly defined. Integration occurs when a subset of replicas implements *more than one* replication protocol, or when rules are established for cross-communication to take advantage of the unique characteristics of a protocol or topology.

We integrate communication at consensus replicas by allowing them to participate in anti-entropy with the eventual cloud (but not with other consensus replicas). Because the consensus replicas are generally a small subset of the overall system, this type of integration ensures that the number of messages in the system does not scale according to the number of replication protocols being federated. Eventually consistent replicas therefore "synchronize" with consensus replicas by exchanging synchronization RPC messages initiated from either the consensus or the EC replica. EC replicas must also reply with failure to consensus RPC messages

(e.g. `AppendEntries` or `VoteRequest`). Failure may indicate that a quorum has changed, requiring joint consensus decisions or a reconfiguration if the consensus group implements hierarchical consensus.

Communication integration can also take advantage of the geographic topology of the system to localize non-broadcast forms of communication. Specifically, EC replicas can prioritize their communication with consensus replicas or local replicas by modifying the random selection of pairwise anti-entropy. In this chapter we propose a policy based approach to peer selection, though in Chapter 6 we propose a learning approach. The policy based approach requires the configuration of two probabilities, $P_{sync}$ and $P_{local}$. During peer selection, the local replica first selects the consistency class of the neighbor, selecting a consensus replica with probability $P_{sync}$, otherwise another EC replica (consensus replicas always select an EC replica). If a consensus replica is selected, then synchronization occurs with the *geographically nearest* available replica. If an EC replica is selected, then a second decision is made between selecting a neighbor in the local area or in the wide area with probability $P_{local}$, at which point a uniform random selection of peers is made.

In the homogeneous eventual case, only $P_{local}$ is relevant. By slightly favoring synchronization and local communication for anti-entropy, the system becomes more reliant on the core consensus group and therefore has stronger global consistency (fewer forks overall). Alternatively, lowering the likelihood of synchronization will allow the system to become less reliant on consensus, particularly when wide area outages are likely.

Varying communication between protocols in this way raises an important

question: does a consensus group improve global consistency because it broadcasts across the wide area, or because it implements a stronger consistency model? We investigated this question by implementing a special eventually consistent replica called the "stentor" replica. Stentor replicas conduct two anti-entropy sessions per replication interval, one across the wide area and one locally. We compared a federation of Raft and eventual with a federation of eventual and stentor and found that while stentor performs slightly better than homogeneous bilateral anti-entropy, consensus has a strong effect on how inconsistencies are handled.

### 4.3.2 Consistency Integration

Consistency integration occurs on communication between replicas with different local consistency policies. When an EC replica receives a synchronization message from a consensus replica it accepts the most recent version. However, the reciprocal consensus operation applies consistency policies such as rejecting forks by initiating a decision with the leader. Forks detected by Federated Raft followers can be dropped without leader interaction, which allows the consensus group to be more available. Per-replica caches of forked and dropped writes are used to detect and prevent duplicate remote updates being sent to the leader (e.g. when the same update is propagated to two followers from the EC fog). Even though a Raft follower notes a fork and does not propagate it, a fork may arrive at another Raft follower that has yet to see it via anti-entropy propagation. Increasing $P_{local}$ can help prevent the EC fog from propagating forks "around the consensus group"

by performing anti-entropy with a consensus replica in another region.

This simple integration alone is not sufficient to improve global consistency, in fact it performs worse than a homogenous system in isolation. The problem is that EC and consensus replicas resolve fork conflicts in exactly opposite ways. EC replicas choose the last of a set of conflicting writes because of the latest writer wins policy, whereas consensus replicas effectively choose the first by dropping any write that conflicts with previously seen updates.

Consider conflicting writes to object $a_1$ by $R_i$ and $R_j$, which create versions $a_{i,2}$ and $a_{j,2}$ ($a_{j,2} > a_{i,2}$ because the precedence id of $R_j$ is greater than that of $R_i$). EC replicas will converge to $a_{j,2}$ because its version is later. However, the consensus replicas will converge to whichever write first reaches the leader, and there is no mechanism by which to override a write that has already been committed. If $a_{i,2}$ is committed by the leader, an impasse is reached and neither write will become fully replicated. This disconnect arises from a fundamental mismatch in the protocols' approaches to conflict resolution, but if we modify either approach, then the protocol will perform less well in a non-federated environment. We resolve this issue by noting that if the strong central quorum can make a write accepted by the consensus replicas "more recent" than any conflicting write, all eventual replicas will converge to the write chosen by consensus.

We therefore extend each version number with an additional monotonically increasing counter called the *forte* (strong) number, which can *only be incremented by the leader of the consensus quorum.* Because the consensus leader drops forks, or any version not more recent than the latest committed version, incrementing the

forte number on commit ensures that only consistent versions have their forte numbers incremented. Version comparison are performed by comparing forte numbers first, and then the conflict free version number, allowing the leader to "bump" its chosen version to a later timestamp than any conflicting writes.

The forte bump must be propagated to derived writes as well to ensure that the branch selected by the consensus group is maintained during synchronization. Otherwise, the increment of an object version forte number would result in child versions derived from the update being erroneously identified as conflicting. On receipt of a version with a higher forte than the local, EC replicas search for the forte entry in their local log, find all children of the update, and set the child version's forte equal to that of the parent.

We believe that the strategy of "nominating the latest write" is sufficient for integrating other consistency protocols as well. At this stage in our investigation, however, there are quite a few parameters that must be tuned, such as timing parameters, policies, and probabilities. We expect that further investigation into smoothing integration points between consistency protocols and policies will lead to fewer RPCs with less messages and processing requirements. The bottleneck in a federated system is the leadership of the central quorum, through which every single update must pass, no matter the size of the system. We address this with hierarchical consensus, which we hope will also provide further opportunities for consistency-centric evaluation.

## 4.4 Performance Evaluation

To investigate the effect of variable latency and the network environment on consistency, we created a discrete event network simulation. Simulating our network environment allowed us to achieve two things. First, a simulation can accurately measure visibility latencies – detecting visibility latency and replication in an actual system is error prone at best and requires a significant amount of logging at worst. Second, the simulated network allowed us to flexibly configure network behavior to test a large range of environments.

### 4.4.1 Discrete Event Simulation

Our simulated network describes a fully connected topology of replicas distributed across several geographic regions as shown in Figure 4.2. Within each region, replicas enjoy stable, low-latency connections with their neighbors. Across regions, the latency is higher and the connections more variable, meaning that out of order messages are more common across the wide area than in the local area. We simulated both replica failures, where a single replica stops responding to messages, and network partitions, where messages can only be exchanged within geographic regions. In both cases, accesses may continue at unresponsive replicas and subnets, though they are not immediately replicated across the system. Partitioned replicas will fall behind the global state, and must be re-integrated into the network when the outage ceases.

Input to the simulation has two parts: a network topology and a workload of

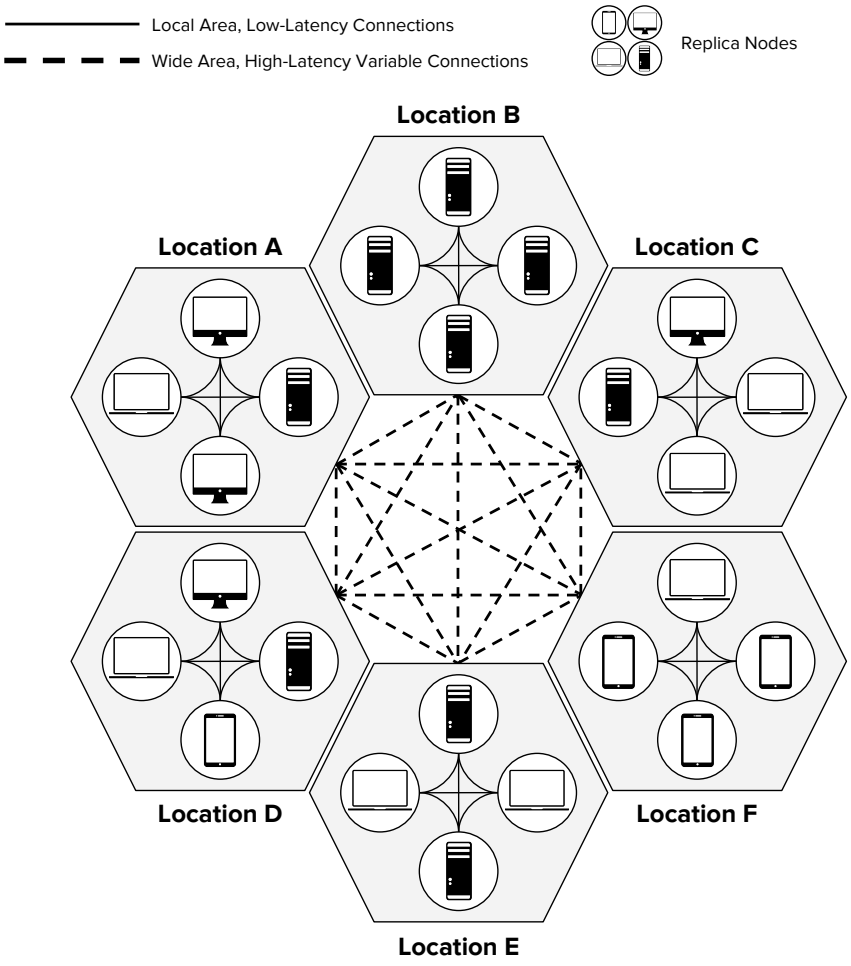


Figure 4.2: We evaluated our federated consistency model in a fully connected simulation. Each simulation specified a topology of replicas that replicas in the same region enjoyed stable, low-latency connections, while across the wide area connections were more variable. Each replica in the system is assigned a different consistency protocol to execute during runtime.

access events. The simulation instantiates each replica as a process that executes read and write accesses to objects, generates replication messages, and handles messages from other replicas. Topologies specify each device as an independent replica by uniquely identifying it with device-specific configurations. By far the most important configuration option is a replica's *consistency* (or replication protocol), which determines the replica's behavior. Our simulation currently defines two types of replicas:

- *Eventual*: Eventual replicas replicate objects with periodic anti-entropy synchronizations. During each synchronization a peer is randomly selected with two selection likelihoods, $P_{sync}$, the probability of synchronizing with a Raft replica and $P_{local}$, the ratio of local vs. wide-area peer selection.

- *Raft*: Raft replicas implement the Raft consensus protocol, electing a leader and forwarding writes to the leader to maintain a sequential ordering of operations. Writes identified as forks of prior committed writes are dropped by whichever replica makes the identification.

The topology further specifies the *location* of each device, the *connections* between devices, and the *distribution* of message latency on a per-connection basis. Each connection defines the latency of messages between replicas, described as a normal distribution ($\lambda_\mu$, $\lambda_\sigma$). There are two basic types of connections: within the local area, or across the wide area. Connections in the local area specify a lower mean latency and less variability than connections between devices in different regions. The tick parameter, $T$, is computed by the average worst-case latency in

the simulation. Each topology also has the capability to set runtime and device-specific configurations, though we do not take advantage of this when presenting these results.

Workloads are specified as access trace files – time-ordered access events (reads and writes) between a specific device and a specific object name. Each trace is constructed with a random workload generator that maps devices to accesses using a distribution of the delay between accesses, the number of objects each replica accesses, $o$, and a probability of conflict, $P_c$. Several distributions are available, including Zipfian and uniform distributions, we are primarily interested in a high-throughput workload, therefore we used a normal distribution of access ($A_\mu$, $A_\sigma$). Object names were assigned to replicas as follows: object names are selected and assigned to replicas, round-robin, with probability $P_c$ until each replica was assigned $o$ objects. If $P_c = 1.0$ then every single replica would access the namespace, whereas if $P_c = 0.0$ then each replica would access a unique set of objects. The access delay distribution was used to generate accesses to objects in sequence, by selecting an object and reading and writing to it over time until some probability of switching objects occurred. In effect, the final workload simulates multiple replicas reading and writing at a moderate pace for approximately one hour.

### 4.4.2 Experiments and Metrics

We conducted two primary experiments to test the behavior of a federated consistency system against homogeneous consensus and EC systems. The first sim-

ulates consistency behavior in the face of increasing failure of wide-area links, in the form of region partitions. The second explores effect of the network environment on consistency as the wide-area $\lambda_\mu$ increases. Our simulated topology consists of twenty replicas distributed across five geographic regions. Eventual replicas prefer to choose replicas within the same geographic region for anti-entropy, e.g. $P_{local}$ is high. Our topology is constructed with an inner core of Raft replicas such that there is one consensus replica per region, colocated with several eventual replicas. Our experiments were condected with synthetic access traces containing approximately 29,000 accesses (depending on the experiment), approximately two thirds of which are reads.

Our primary metrics are *stale reads* and *forked writes*, which produce application-visible effects. We define forked writes as the number of writes that had more than one child (multiple writes to the same parent version), whereas reads are stale if they return anything other than the globally latest version. We also measured *write visibility*. Recall that a write is *visible* if and when it is propagated to all replicas. Any writes that do not become fully visible (e.g. are stomped as they are propagated through the EC dissemination network) are ignored. This metric is closely related to the *percent visible* metric – the average number of replicas a write is propagated to. These metrics are all made possible in a simulated environment, measuring these effects in a real geo-replicated system would require a great deal of logging and data post-processing which would be error-prone due to clock synchronization errors.

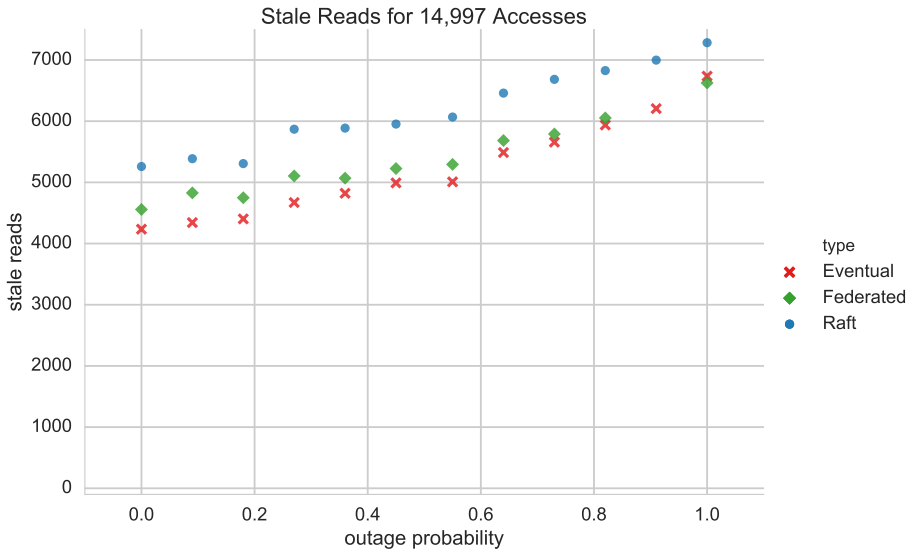


Figure 4.3: Stale reads as the probability of wide-area outages increases.

### 4.4.3 Wide Area Outages

Our first simulation experiments considered the effect of outages that partitioned each simulated region so that they could not communicate with each other. Each simulation was parameterized by a probability of failure, $P_f \in [0.0, 1.0]$ The $P_f$ was used at runtime to determine if an outage was going to occur at any given timestep and the length of the outage. A $P_f = 0.5$ indicates that all wide-area links are simultaneously down (messages cannot be sent across the wide-area) 50% of the time, whereas a $P_f = 1.0$ indicates that all wide-area links are permanently down after a short initial online duration. In the case of an outage, anti-entropy synchronizations will proceed as usual within a region, however Raft will become unavailable as every replica becomes a candidate for the duration of the outage.

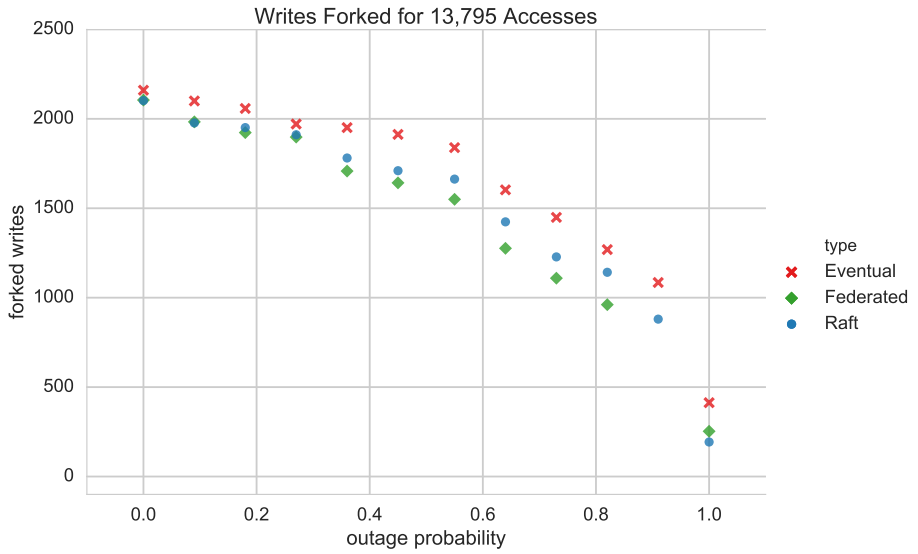


Figure 4.4: Forked writes as the probability of wide-area outages increases.

We show the effect of outages on consistency by measuring stale reads in Figure 4.3 and forked writes in Figure 4.4. The eventual replicas deal with increasingly poor network conditions the best: randomized anti-entropy partner selection allows writes to propagate through multiple paths. Anti-entropy and eventually consistent systems are widely used precisely because of their ability to remain highly-available during network outages. When federated, the system is able to leverage the eventual subset of its replicas to route around failures almost as efficiently as the homogeneous Eventual system.

The multiple-paths ability also allows the federated system to propagate writes quickly, as shown in Figure 4.4. In fact, the federated system outperforms a homogenous eventual system, possibly because the Raft quorum is able to quickly disseminate writes during those periods when wide-area links are available.

These experiments considered complete network partitions as the failure mode, however, if the failure mode was instead random replica failures, the system would respond differently because of the way we configured the central quorum. In a quorum size of 5, Raft can handle 2 failures before an extended outage. Leader failure would cause temporary outages until the election timeout occurs, but would be online with only a minimum of missed accesses. If a Raft replica fails inside of a region, the eventually consistent replicas in that region can still make progress without the quorum. The system also maintains the benefits of the core consensus group as synchronizations across the wide area may find a Raft replica. Updates would still propagate across the wide area without a central broadcast mechanism at the cost of an increased number of forks as reads become increasingly stale without a local quorum component.

### 4.4.4 Latency Variability

Our second experiment investigates the effect of variable network latency on consistency protocols and how the selection of the tick parameter model affects consistency for each system. Each simulation is parameterized by a $T$ parameter that is a function of the wide area $\lambda_\mu$ and $\lambda_\sigma$. We used the same workload trace across all simulations, fixing the access mean, $A_\mu = 3000$, e.g. approximately one access per replica every 3 seconds. In effect, this meant that for approximately half the simulations (with higher latencies), it was impossible for a write to become visible on another replica before a fork. The probability of conflict for the workload

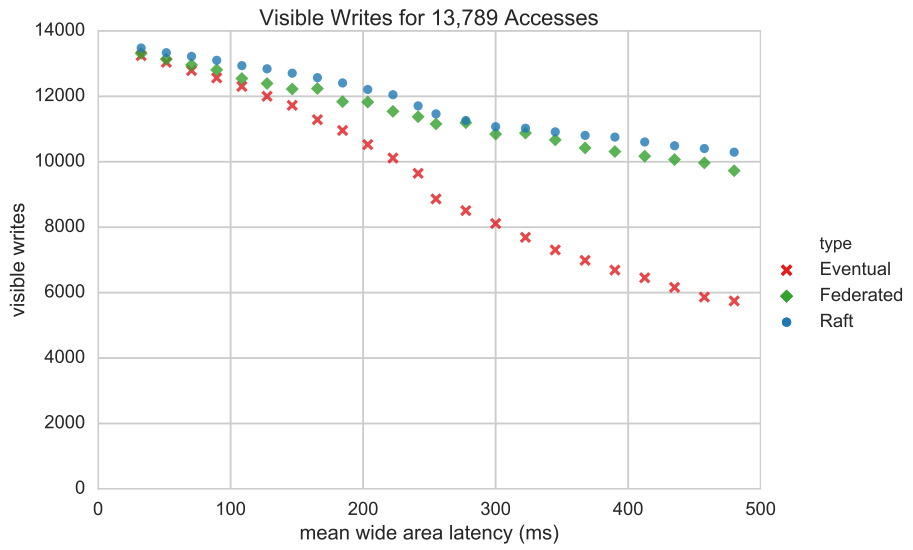


Figure 4.5: The percentage of fully visible writes as the mean wide area latency increases.

was set to $P_c = 0.5$, however, there was still enough conflict due to connection latencies to force each protocol to handle many forks.

Figures 4.5 and 4.6 show that write propagation is much faster and more effective in Raft than in eventual, especially as network conditions deteriorate. Raft ensures that writes become fully replicated at the cost of increased write latencies, moreover they require broadcast over the wide area. This is not an ideal scenario in failure prone networks, but broadcast from a single leader ensures propagation is fast. Federated essentially splits the difference between Raft and eventual in terms of mean replication latency. However, Figure 4.5 shows that federated fully replicates many more writes than eventual, closely tracking the number of writes fully replicated by Raft.

The strong inner core of Raft replicas is the key to the federated protocol

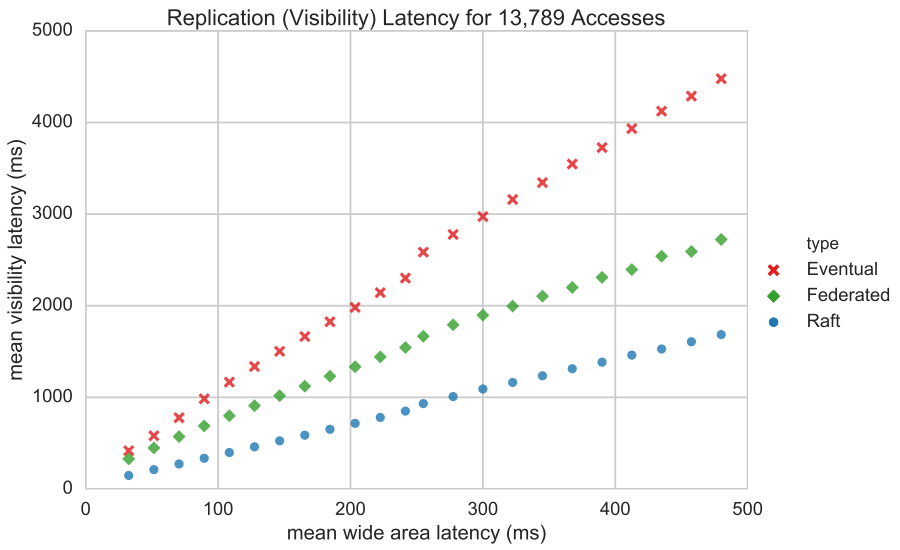


Figure 4.6: The average amount of time an update becomes fully visible (if the update becomes fully visible).

tracking Raft's performance. EC replicas are biased in favor of performing anti-entropy with local replicas, allowing most anti-entropy sessions to perform quickly and without delay. By contrast, the Raft replicas in the federated topology are intentionally spread across geographic regions. A new write originating at an eventual replica is quickly spread to the local Raft replica, and is then broadcast to the rest of the regions via consensus decisions. Disseminating writes quickly minimizes the possibility of another, later eventual write starting up concurrently. Additionally, the forte number prevents new forked writes from stomping on a conflicting write disseminated via Raft replicas.

The effect of Raft disseminating updates across the wide area can be seen in access network extracted from our simulation shown in Figure 4.7. In this network, vertices represent replicas and are colored by the consistency protocol it implements

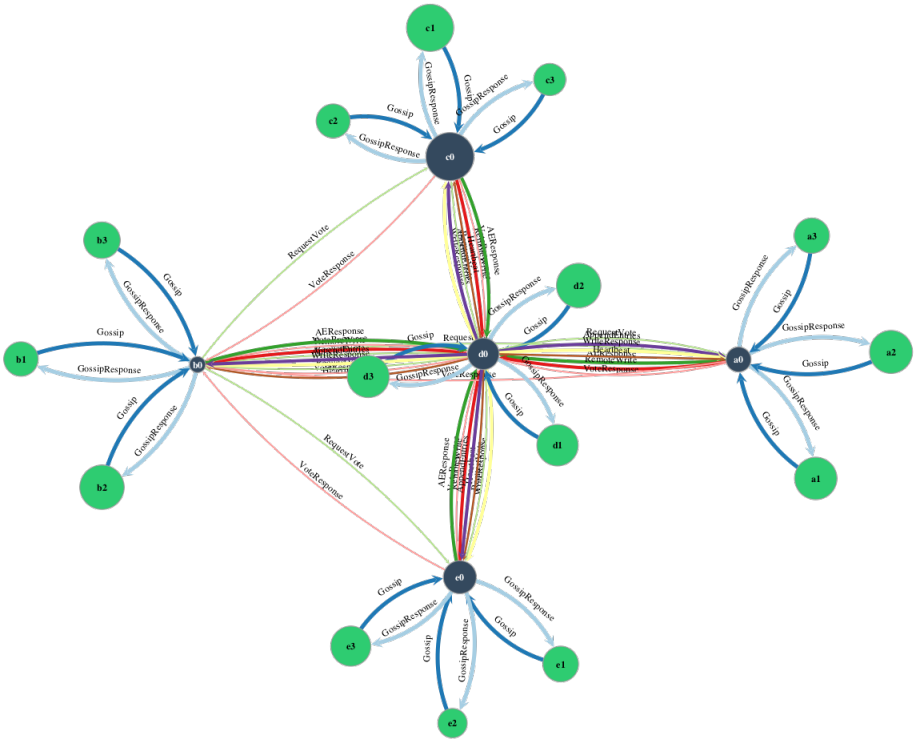


Figure 4.7: This graph shows the synchronization of a federated topology for a simulation run that optimizes Raft connections in the wide area and eventual connections in the local area. Vertex size indicates the number of accesses at each replica and color represents the replica type (blue for Raft, green for eventual). Edges are colored by RPC type and sized by the number of messages sent.

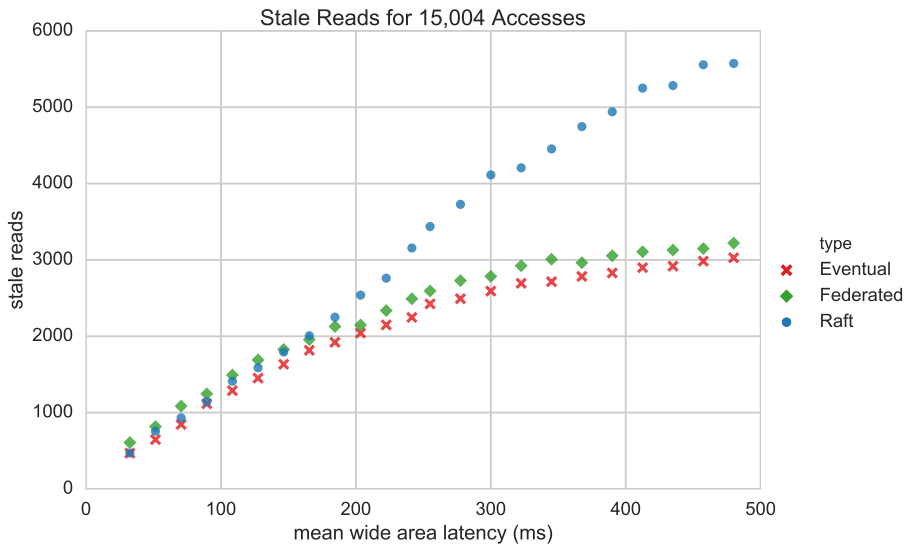


Figure 4.8: The total number of reads that are stale when they are executed.

(blue for Raft, green for eventual). The size of the vertex represents the number of accesses that occur at each location. The edges are colored by RPC type and are sized by the number of message of that type sent between replicas. This network shows an extreme optimization of a federated network, using Raft as a broadcast network across the wide area and local synchronization to anti-entropy nodes. Although this type of network did not perform as well in the outage simulations, it performed very well for our variable latency simulations. These observations served as the basis for our planetary architecture as described in § 2.2.3.

Figures 4.8 and 4.9 show the average number of stale reads and forked writes across different mean latencies. All three protocols perform similarly at smaller latencies, but eventual and federated deal with high latencies much more effectively than Raft, at least for this size of system.

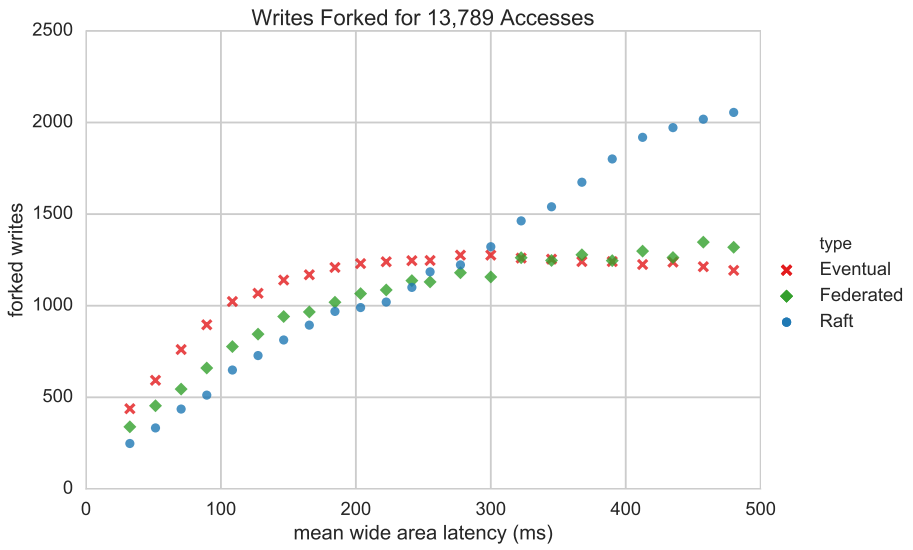


Figure 4.9: The total number of writes that are forked, potential application-level conflicts.

Higher latencies affect Raft in at least two ways. First, higher latency variability causes more out of order messages. Second, system timeouts are parameterized by $T$ which, in turn, is based on mean latencies. The result is that Raft's append entries delay is longer for simulations with higher mean latencies, resulting in more conflicts. The same is true for anti-entropy delays, but the speed of Raft decisions is determined by the slowest quorum member, which can be quite slow when message variability is large. By contrast, a slow anti-entropy participant only affects direct anti-entropy partners, not the replication of the update across the whole system.

Though not shown here, we also investigated the effect of changing the number of replicas in the system (a system implementation shows that consensus does not scale well in Figure 3.9). As system size increases, more time is required to fully replicate writes, increasing the likelihood of both stale reads and forks. Equation 4.1

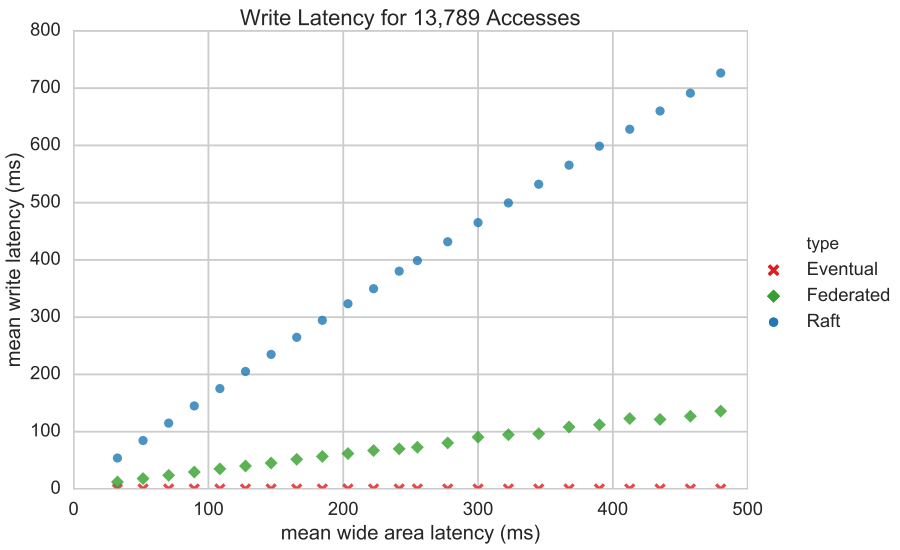


Figure 4.10: Average cost of writes to local replicas. Eventual consistency writes are completed immediately without coordination, therefore have zero cost. The cost of Raft depends on the latency of the majority vote. Federated with more eventual replicas will more than halve the average write cost in the system.

shows that bilateral anti-entropy propagates writes to $N$ nodes exponentially. Given the relationship of the `anti-entropy delay` and `heartbeat interval` expressed by $T$, Raft broadcasts overtake anti-entropy between 9 (2 anti-entropy sessions) and 27 replicas (3 anti-entropy sessions).

Finally, Figure 4.10 shows the mean synchronous cost of write operations to ongoing computations. All Raft writes must be forwarded to the leader to be serialized and are completed after a round trip communication. Eventual writes are local and are completed immediately, even before being replicated, and therefore have zero write cost. Most federated replicas are eventual and therefore federated's average write cost tracks eventual's relatively closely.

## 4.5 Conclusion

In this chapter we have presented a model for federated consistency to hybridize consistency protocols in a single system. Federated consistency allows individual replicas to expose local consistency policies to users, while still allowing for global guarantees. We explored the federation of eventual consistency implemented with anti-entropy synchronization and sequential consistency implemented with Raft consensus.

Federation requires both communication and consistency integration at the consistency boundary, that is when replicas of different policies interact. We solved communication integration by identifying how each consensus protocol should respond to RPCs of the other, and by introducing parameters that modified how peers were selected to communicate with each other. Consistency integration involved ensuring that decisions made by either protocol were respected by the other. By default this is not the case, since each protocol selected writes in opposite ways. We therefore had to have a way for each protocol to determine the most relevant write to propagate, which we solved by extending conflict-free version numbers with a forte number that could only be incremented by the Raft leader. Though we only investigated eventual and sequential consistency, we propose that other consistency models, e.g. causal consistency, could be similarly federated.

We evaluated federated consistency in the context of a geographically dispersed wide-area object store using a simulation to track metrics not generally available in a real implementation. Our results show that a key to the global guarantees is

using a core consensus group to serialize and broadcast system writes. By designing a federated system where only interactions between replicas of varying consistency types are defined, systems can scale beyond the handful of devices usually described to dozens or hundreds of replicas in variable-latency, partition-prone geographic networks. Replicas can monitor their local environment and adapt as necessary to meet timeliness and correctness constraints required by the local user.

We were only able to investigate a limited number of system configurations and the space of possible system configurations is vast. We do not claim that the configurations described in this chapter are in any way optimal. Rather, we claim that our simulation results described in § 4.4 show that the general approach is promising. Our simulation environment is extremely flexible, and we intend to continue evaluating possible system configurations in parallel with our system development.

Federated consistency has the potential to scale system sizes to extremely large networks of millions of nodes. For this to happen our ideal configuration using a central Raft quorum must also scale, which is possible when using hierarchical consensus. Planetary scale systems comprised of a fog of highly available, eventually consistent replicas federated with a central core of strong consistency hierarchical consensus will allow high throughput, rapid replication, high availability, and resistance to outages and variable network conditions.

# Chapter 5: System Implementation

Given its grandiose title, it may seem that the engineering behind the development of a planetary scale data storage system would require thousands of man-hours of professional software engineers and a highly structured development process. In fact, this is not necessarily the case for two reasons. First, data systems benefit from an existing global network topology and commercial frameworks for deploying applications. This means that both the foundation and motivation for creating large geo-replicated systems exists, as described in Chapter 2. Second, like the internet, complex global systems emerge through the composition of many simpler components following straight forward rules [147]. Instead of architecting a monolithic system, the design process is decomposed to reasoning about the behavior of single processes.

Fundamentally, each process in our system is an independent actor with storage, memory, and compute resources [148–150]. The primary purpose of an actor is to receive and respond to messages (events) from other actors by modifying the actor's internal state, creating new actors, or sending messages to other actors [151]. The behavior of an actor depends solely on the order of messages received, making them an ideal model for programming consistency protocols. A system is therefore

composed of actors, and reasoning about the global behavior of the system requires only a description of the interactions that different actors in the system have.

The actor model allows us to decouple consistency behavior from application behavior. Consistency behavior is defined in the messaging between actors, for example actors participating in hierarchical consensus provide consistency by voting for a leader and correctly committing commands from the leader based on majority votes. Application behavior is defined by the internal state of the actor, for example the maintenance of a versioned key-value store. We use this decoupling to construct three principle applications from our consistency-centric model: a ledger, a key-value database, and a file system, all distributed geographically.

In this chapter we will describe the details of our implementation. First, we will describe the base requirements for all replicas, along with our assumptions concerning communication, security, processing and data storage. We will also outline the details of our implementation of the consistency protocols described in previous chapters. Finally we will describe the details of the applications we built on top of our consistency protocols.

## 5.1 Replicas

The primary actor in our system is the *replica*. Replicas are independent processes that maintains a portion of the objects stored as well as a *view* of the state of the entire system. Each replica implements a shared-nothing architecture [152] such that each replica has its own memory and disk space. For practical purposes of

fault tolerance, we generally assume that there is a one-to-one relationship between a replica and a disk so that a disk failure means only a single replica failure. Replicas must be able to communicate with one another and may also *serve* requests from clients. By default we assume that all replicas in the system are addressable and that both clients and peers can send messages to all replicas in the network, barring failures.

A system is composed of multiple communicating replicas and is defined by the behavior the replicas. For example, a totally replicated system is one where each replica stores a complete copy of all objects as in the primary-backup approach [153], whereas a partially replicated system ensures durability such that multiple replicas store the same object but not all replicas store all objects as in the Google File System (GFS) [65]. At the scale of a multi-region, globally deployed system, we assume that total replication is impractical and primarily consider the partial replication case. However, we also assume that replicas maintain a view of the entire system, that is meta-data about the location and provenance of all objects, so as to direct client requests to the appropriate replica to serve requests.

We assume that replicas reside on trusted hosts with reliable communication and that failures are non-byzantine. However, we do expect security to be a default component of a real-world implementation, particularly as many communication links travel across the internet. Communication should be secured and authenticated with transport layer security (TLS) [154, 155]. TLS requires that each replica maintains a certificate and public key encryption to secure communications [156] and if each replica has its own certificate, then TLS can also be used to authenticate

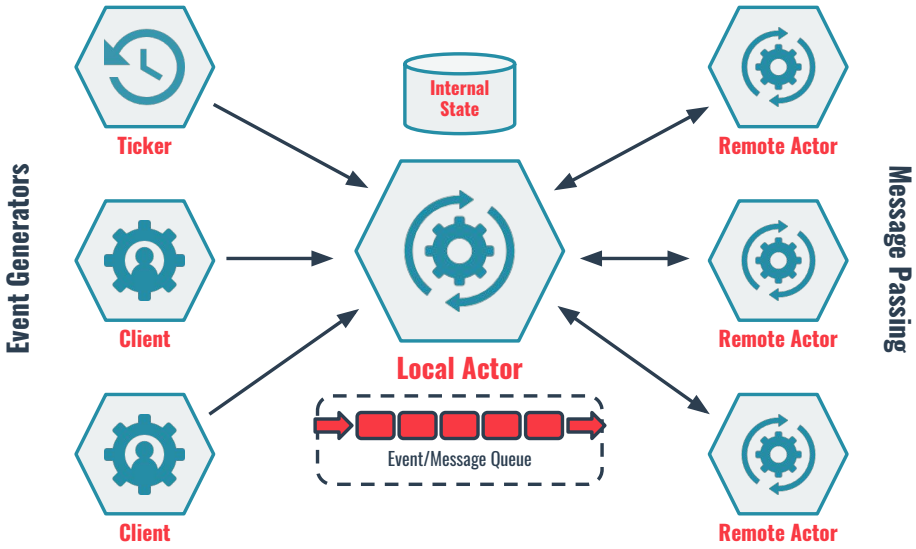


Figure 5.1: Each replica is an actor that maintains an internal state that is modified by processing messages from other actors. Implemented in the Go programming language, each event is serialized by a single message channel ensuring that the state is modified correctly.

valid peers based on a central authority's shared certificate [157]. We also assume that data stored on disk should be encrypted. We prefer per-replica encryption to ensure that data is loaded and stored from an in-memory cache as quickly as possible and though we recognize that some applications require per-user encryption, it is beyond the scope of our system to provide it.

All replicas are implemented in Go [158], a systems programming language that provides concurrency through communication channels [159]. Each replica implements a primary event loop as a single channel to ensure that all events and messages are serialized in a single order as shown in Figure 5.1. This prevents the need to use multiple expensive mutexes to synchronize the behavior of multiple

threads and allows us to more easily reason about the operation of the system. Communications are implemented using gRPC [160], an HTTP communication protocol that serializes messages in the protocol buffers format [161] which allows clients to be implemented in multiple programming languages. The gRPC server accepts new requests each in their own thread. Clients are handled using unary RPC requests, but to improve communication performance between replicas we use bilateral streaming. Each message is pushed through the primary event channel, then responded to using a callback channel. Other threads include timers and monitoring threads that are also synchronized through the main event channel.

We've principally implemented two types of replicas based on consistency protocol. Alia [162] replicas implement hierarchical consensus based on our implementation of the Raft protocol [163]. Honu [164] replicas implement eventual consistency using bilateral anti-entropy synchronization. Both Alia and Honu implement an object store such that objects are described by unique keys and each update to the object creates a new version. The key/value nature of our implementation allows the namespace and object data to be sharded and partially replicated across all replicas, therefore the key/value database described in § 5.2.1 is the base application for all of our applications.

Replicas primarily cache their object stores in memory to improve performance. If a replica fails it can be brought up to date by a peer replica through either consistency protocol. Durable storage is written to asynchronously, to minimize the amount of recovery time required for a replica. Both Alia and Honu can use multiple backend stores, writing pages and logs to disk, or using embedded

key/value stores such as LevelDB [165], BadgerDB [166], or PebblesDB [167]. These databases use the ext4 file system, though in the future we hope to investigate the use of a tree-based file system to more directly optimize disk usage [168].

### 5.1.1 Alia

Hierarchical Consensus with modified Raft as the underlying consensus protocol exports a linearizable order of accesses to the distributed key-value store. Alia and the HC library are implemented in Golang use gRPC [160] for communication. The system is implemented in 7,924 lines of code, not including standard libraries or support packages.

An replica implements multiple instantiations of the Raft protocol, which we have modified in several ways. Every replica must run one instantiation of the *root consensus protocol*. Replicas may also run one or more instantiations of the *commit consensus protocol* if they are assigned to a subquorum. Repartition decisions move the system between epochs with a new configuration and tagspace, and can only be initiated by messages from peers or monitoring processes. A successful repartition results in a new epoch, tagspace, and subquorum topology committed to the root log. `Repartition` messages also serve to notify the network about events that do not require an epoch change, such as the election of a new subquorum leader or bringing a failed node back online.

Each replica implements an event loop that responds to timing events, client requests, and messages from peers. Events may cause the replica to change state,

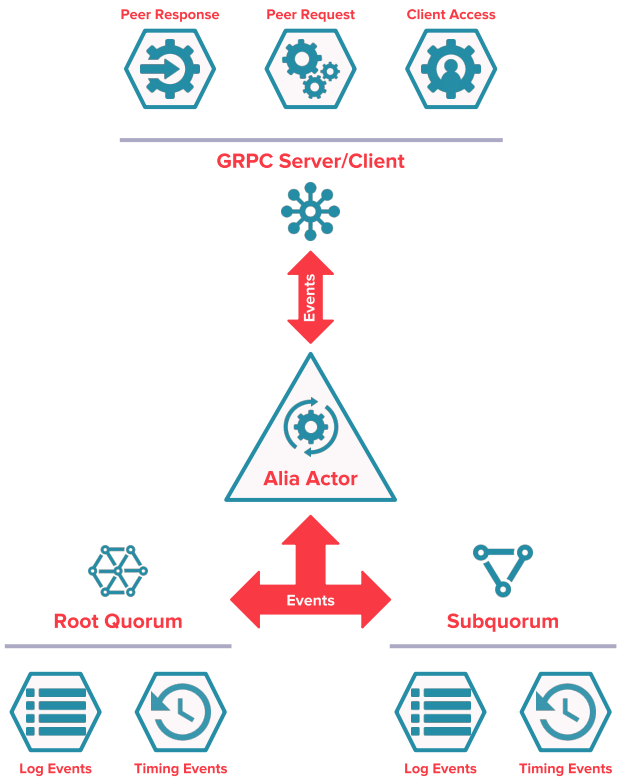


Figure 5.2: The Alia actor model model is composed of at least three primary actors: the root quorum actor, a subquorum actor, and the main actor that dispatches messages to each subquorum. Other minor actors such as client requests threads and timing events dispatch their messages directly to their associated actors.

modify a command log, broadcast messages to peers, modify the key-value store, or respond to a client. Event handlers need to aggressively lock shared state for correctness because Golang and gRPC make extensive use of multi-threading. The balance between correctness and concurrency-driven performance leads to increasing complexity and tighter coupling between components, one that foreshadows extra-process consistency concerns that have been noted in other work [86, 92, 169].

The computing and network environment of a distributed system plays a large role in determining not just the performance of the system, but also its behavior. A simple example is the election timeout parameter of the Raft consensus protocol, which must be much greater than the average time to broadcast and receive responses, and much less than the mean time between failures [59, 92, 170]. If this requirement is not met, leader may be displaced before heartbeat messages arrive, or the system will be unable to recover when a leader fails. As a result, the relationship between timeouts is critically dependent on the mean latency ($\lambda_\mu$) of the network. Howard [146] proposes $T = \lambda_\mu + 2\lambda_\sigma$ to determine timeouts based on the distribution of observed latencies, sets the heartbeat as $\frac{T}{2}$, and the election timeout as the interval $U(T, 2T)$. We parameterize our timeouts (Table 5.1) on latency measurements made before we ran our experiments. Monitoring and adapting to network conditions is part of ongoing work.

**Changes to base Raft:** In addition to major changes, such allowing replicas to be part of multiple quorums simultaneously, we also made many smaller changes that had pervasive effects. One change was including the *epoch* number alongside the term in all log entries. The epoch is evaluated for invariants such as whether or

Table 5.1: Parameterized timeouts in our implementation. The *obligation* timeout stops a partitioned subquorum after an extended time without contact to the rest of the system. $T = 10msec$ for our experiments on Amazon EC2.

| Name | Time | Actions |
|---|---|---|
| sub heartbeat<br>sub leader | 1T<br>2-4T | sub leader heartbeat<br>new sub election |
| root heartbeat<br>root election | 10T<br>20-40T | root leader heartbeat<br>new root election |
| obligation | 50T | root quorum may re-allocate the tag |

not a replica can append an entry or if a log is as up to date as a remote log.

Vote delegation requires changes to vote counting. Since our root quorum membership actually consists of the entire system, all replicas are messaged during root events. All replicas reply, though most with a "zero votes" acknowledgment. The root uses observed vote distributions to inform the ordering of future consensus messages (sending requests first to replicas with votes to cast), and uses timeouts to move non-responsive replicas into "hot spares" status.

We allow `AppendEntries` requests in subquorums to aggregate multiple client requests into a single consensus round. Such requests are collected while an outstanding commit round is ongoing, then sent together when that round completes. The root quorum also aggregates all requests within a minimum interval into a single new epoch-change/reconfiguration operation to minimize disruption.

Commits are observed by the leader once a majority of replicas respond positively. Other replicas learn about the commit only on the next message or heartbeat. Root epoch changes and heartbeats are designed to be rare, meaning that epoch

change commits are not seen promptly. We modified the root protocol to inform subquorums of the change by sending an additional heartbeat immediately after it observes a commit.

Replicas may be part of both a subquorum and the root quorum, and across epoch boundaries may be part of multiple subquorums. In principle, a high performance replica may participate in any number of subquorums. We therefore allow replicas to accommodate multiple distinct logs with different access characteristics.

Peers that are either slow or with unsteady connectivity are occasionally left behind at subquorum leader or epoch changes. Root heartbeats containing the current system configuration are broadcast to all replicas and serve to bring them up to date.

Finally, consensus protocols often synchronously write state to disk before responding to remote requests. This allows replicas that merely crash to reboot and rejoin the ongoing computation after recovering state from disk. Otherwise, these replicas need to go through heavyweight leave-and-rejoin handshakes. Our system avoids these synchronous writes by allowing epochs to re-join a subquorum at the next epoch change without any saved state, avoiding these handshakes altogether.

### 5.1.2 Honu

## 5.2 Applications

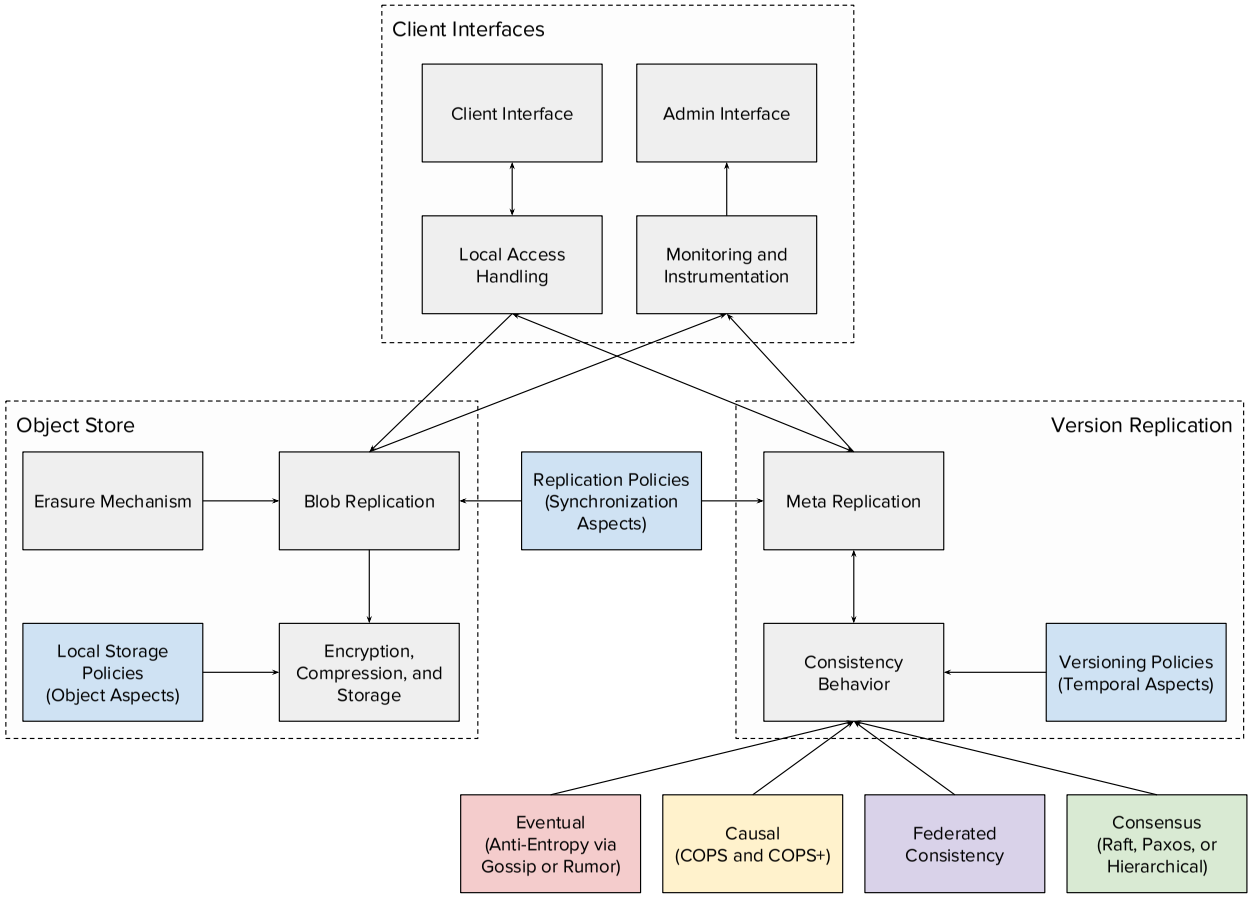


Figure 5.3: This architecture provides a general component model for consistency-centric applications. Object blobs and version metadata are replicated separately to allow partial replication of data but a full view of the current state of the system. Only version replication requires consistency semantics, therefore our federated and hierarchical consensus models only deal with metadata commands.

### 5.2.1 Key-Value Database

### 5.2.2 File System

In the context of a wide-area file system, those operations could be individual `write()` systems calls, though this would be inefficient. Most wide-area file systems aggregate individual accesses through *Close-To-Open* (CTO) consistency, where file reads and writes are "whole file" [171–173]. A file read ("open") is guaranteed to see data written by the latest write ("close"). This approach satisfies two of the major tenets of session consistency: `read-your-writes` and `monotonic-writes`, but not `writes-follow-reads` [74, 75, 78].

Our file system, like many modern file systems, decouples meta-data *recipes* [65, 174–177] from file data storage. Meta-data includes an ordered list of *blobs*, which are opaque binary chunks. When a file is closed after editing, the data associated with the file is *chunked* into a series of variable-length blobs [173], identified by a hashing function applied to the data [178, 179]. Since blobs are effectively immutable [180], or tamper-evident, (blobs are named by hashes of their contents), we assert that consistent meta-data replication can be decoupled from blob replication. Accesses to file system meta-data becomes the operations or entries in replicated logs. Meta-data is therefore replicated through the system, allowing any file system client to have a complete view of the file system namespace, even while not caching any file data.

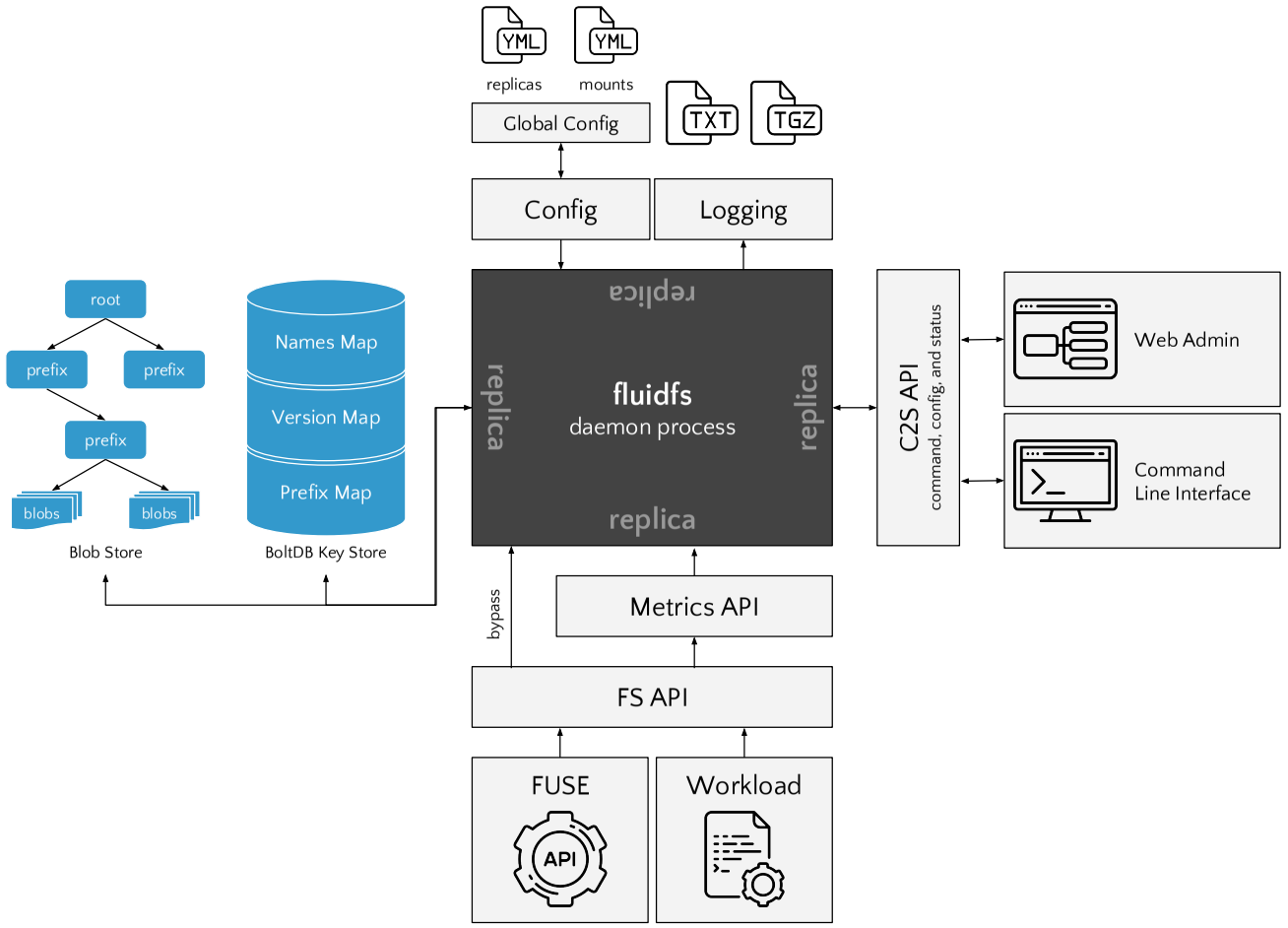


Figure 5.4: FluidFS implements the application component model specifically for a file system. Users interact with a local replica that implements the FUSE API or more directly through a gRPC API. The FluidFS daemon process stores data in a blob store and meta data in an embedded key/value store. The process also provides configuration and logging as well as a web interface for command and control.

### 5.2.3 Distributed Ledger

## 5.3 Conclusion

We did not optimize our research for the minimum set of assumptions required to facilitate interoperability between heterogeneous replicas. However, we hope that the assumptions we did make shed light on what is required to achieve the minimum set of assumptions.

Further research is required to ...

# Chapter 6: Adaptive Consistency

Throughout this dissertation we've outlined a planetary-scale data storage system composed of a two-tier structure that provides a hybrid consistency model. Both tiers are designed to scale to thousands of replicas, and together could represent millions of replicas operating in concert around the world. Management and systems administration of such a large scale system using external monitoring processes is impractical at best and prohibitively complex at worst. Even with a trusted infrastructure of cloud services, building a single synchronization point for monitoring and optimization would require the online collection of live information from across the globe. This synchronization point would itself be susceptible to delays and partitions and would have to manage a huge number of events streaming in from a large number of sources, which has challenges in and of itself [181].

Instead, we propose that an *emergent model* of network behavior is required to tune and optimize planetary scale systems in an online fashion such that local, simple rules lead to globally emergent behavior [182]. Specifically we hypothesize that when individual replicas follow simple optimization procedures based on monitoring of their local network performance, access patterns, queries to their neighbors, and other environmental factors the performance of the system will collectively increase.

Because we focus primarily on the consistency aspects of geo-replicated data storage, we have termed this behavior *adaptive consistency*, because with a hybrid or continuous consistency model, such optimizations will minimize inconsistent behaviors due to latency or configuration.

In this chapter we will show we can improve consistency of the system as a whole with localized machine learning implemented on a per-replica basis. Although this work is largely left for future research on a fully deployed platform, we have built our system with this kind of adaptation in mind. Our preliminary experiments suggest that adapting anti-entropy selection with reinforcement learning techniques will meaningfully enhance consistency in the federated fog layer of the system [183]. We will then finish with a discussion of how we can generalize this process to other techniques in the system as a whole.

## 6.1 Anti-Entropy Bandits

A distributed system is made highly available when individual servers are allowed to operate independently without failure-prone, high latency coordination. The independent nature of the server's behavior means that it can immediately respond to client requests, but that it does so from a limited, local perspective which may be inconsistent with another server's response. If individual servers in a system were allowed to remain wholly independent, individual requests from clients to different servers would create a lack of order or predictability, a gradual decline into inconsistency, i.e. the system would experience *entropy*. To combat the effect of

entropy while still remaining highly available, servers engage in periodic background *anti-entropy sessions* [117].

Anti-entropy sessions synchronize the state between servers ensuring that, at least briefly, the local state is consistent with a portion of the global state of the system. If all servers engage in anti-entropy sessions, the system is able to make some reasonable guarantees about consistent replication; the most famous of which is that without requests the system will become globally consistent, eventually [78]. More specifically, inconsistencies in the form of stale reads can be bound by likelihoods that are informed by the latency of anti-entropy sessions and the size of the system [93, 94]. Said another way, overall consistency is improved in an eventually consistent system by decreasing the likelihood of a stale read, which is tuned by improving the *visibility latency* of a write, the speed at which a write is propagated to a significant portion of servers. This idea has led many system designers to decide that eventual consistency is "consistent enough" [11, 12], particularly in a data center context where visibility latency is far below the rate of client requests, leading to practically strong consistency.

However, propagation rates need to be re-evaluated when replicas move outside of data center contexts and when anti-entropy is replicating across the wide area. Our system envisions a fog layer that provides data services to localized regions with a hybrid consistency model. The fog is specifically designed to handle mobile users, sensor systems, and high throughput applications at the edge of the data center backbone [123, 184]. However, scaling an eventually consistent system to dozens or even hundreds of nodes increases the radius of the network, which leads

to increased noise during anti-entropy e.g. the possibility that an anti-entropy session will be between two already synchronized nodes. Geographic distribution and extra-datacenter networks also increase the latency of anti-entropy sessions so that inconsistencies become more apparent to external observers.

To address this challenge, we propose the use of reinforcement learning techniques to optimize network behavior to minimize latency. Anti-entropy uses gossip and rumor spreading to propagate updates deterministically without saturating the network even in the face of network outages [141, 185, 186]. These protocols use uniform random selection to choose synchronization peers, which means that a write occurring at one replica is not efficiently propagated across the network. In this section we explore the use of *multi-armed bandit* algorithms [187, 188] to optimize for fast, successful synchronizations by modifying peer selection probabilities. The result is a synchronization topology that emerges according to access patterns and network latencies. As we will show in the next sections, such topologies produce efficient synchronization, localize most data exchanges, lower visibility latency, and increase consistency.

### 6.1.1 Visibility Latency

In this section we review the access and consistency model in the context of bandits as well as how anti-entropy is conducted. A more complete discussion of these topics can be found in Chapter 4.

Clients can `Put` (write) and `Get` (read) key-value pairs to and from one or

more replicas in a single operation, creating read and write quorums that improve consistency by enforcing coordination between replicas on the access. In large, geo-replicated systems, we assume that clients prefer to choose fewer, local replicas to connect with, assuming that writes are primarily local and reads are global. On `Put`, a new conflict-free version of the write is created. This results in the possibility of two types of inconsistencies that occur during concurrent accesses: stale reads and forked writes. As a write is propagated through the system, the latest-writer wins policy means that at least one of the forks will be "stomped", e.g. not fully replicated.

Both forms of inconsistency can be primarily attributed to *visibility latency*, that is the time it takes for an update to propagate to all replicas in the system. Visibility latency is directly related to the likelihood of stale reads with respect to the frequency of accesses [94]; said another way, decreasing the visibility latency improves the overall consistency of a system. However, in a system that uses anti-entropy for replication, the propagation speed of an update is not governed solely by network connections, it is also bound to the number and frequency of anti-entropy sessions conducted as well as the radius of the network.

Visibility latency is minimized when all replicas choose a remote synchronization partner that does not yet have the update. This means that minimal visibility latency is equal to $t \log_3 n$, where $t$ is the anti-entropy interval and $n$ is the number of replicas in the network. In practice, however, because of inefficient exchanges due to uniform random selection of synchronization partners, this latency is never practically achieved, and is instead modulated by a noise variable that is proportional

to the size of the network.

### 6.1.2 Multi-Armed Bandits

To combat the effect of noise on visibility latency our initial approach employs a technique commonly used in active and reinforcement learning: multi-armed bandits. Multi-armed bandits refer to a statistical optimization procedure that is designed to find the optimal payout of several choices that each have different probabilities of reward. In this case, we use bandits to improve uniform random selection of peers so that replicas choose synchronization partners that are most likely to exchange information, and thus more quickly propagate updates, while still maintaining the properties of full replication and fault tolerance.

A bandit problem is designed by identifying several (usually more than two) competing choices called "arms" [1] , as well as a reward function that determines how successful the selection of an arm is. During operation, the bandit selects an arm, observes the rewards, then updates the payout likelihood of the selected arm, normalized by the number of selections. As the bandit selects arms, it learns which arm or arms have the highest likelihood of reward, and can modify it's arm selection *strategy* to maximize the total reward over time.

Bandits must balance exploration of new arms with possibly better reward values and exploitation of an arm that has higher rewards than others. In the *epsilon greedy strategy*, the bandit will select the arm with the best reward with some

[1] Arms refer to the pulling mechanism of a slot machine, the metaphor generally used to motivate the multi-armed bandit problem.

probability $1 - \epsilon$, otherwise it will select any of the arms with uniform probability. The smaller $\epsilon$ is, the more the bandit favors exploitation of known good arms, the larger $\epsilon$ is, the more it favors exploration. If $\epsilon = 1$ then the algorithm is simply uniform random selection. A simple extension of this is a strategy called *annealing epsilon greedy*, which starts with a large $\epsilon$, then as the number of trials increases, steadily decreases $\epsilon$ on a logarithmic scale. There are many other bandit strategies but we have chosen these two simple strategies for our initial research to demonstrate a bolt-on effective improvement to existing systems.

Peer selection for anti-entropy is usually conducted with uniform random selection to guarantee complete replication. To extend anti-entropy with bandits, we design a selection method whose arms are remote peers and whose rewards are determined by the success of synchronization. The goal of adding bandits to anti-entropy is to optimize selection of peers such that the visibility latency becomes closer to the optimal propagation time as a synchronization topology emerges from the bandits. A secondary goal is to minimize anti-entropy latency by preferring local (in the same data center) and regional (e.g. on the same continent) connections.

Our initial reward function favors synchronizations to replicas where the most writes are occurring by giving higher rewards to anti-entropy sessions that exchange later versions in either a push or a pull, as well as additional rewards if more than one object is exchanged. Additionally, the latency of the synchronization RPCs is computed to reward replicas that are near each other. The complete reward function is given in Table 6.1: for each phase of synchronization (push and pull), compute the reward as the sum of the propositions given. For example if a synchronization results

in three objects being pulled in 250ms, and one object being pushed in 250ms, the reward is 0.75.

Table 6.1: The rewards function for our initial anti-entropy bandits. Rewards are computed by introspecting the results of the pull and push phases of bilateral anti-entropy.

| | **Pull** | **Push** | **Total** |
|---|---|---|---|
| Synchronize at least 1 object | 0.25 | 0.25 | 0.50 |
| Additional for multiple objects | 0.05 | 0.05 | 0.10 |
| Latency $\leq$ 5ms (local) | 0.10 | 0.10 | 0.20 |
| Latency $\leq$ 100ms (regional) | 0.10 | 0.10 | 0.20 |
| *Total* | *0.50* | *0.50* | *1.00* |

The design of reward functions can be implemented to the needs of a specific system. For example, in a system that has workloads with variable sized writes, object size could be considered or systems with imbalanced deployments might consider a reward function that prioritizes inter-region communication.

### 6.1.3 Experiments

We conducted experiments using a distributed key-value store totally replicated across 45 replicas in 15 geographic regions on 5 continents around the world. Replicas were hosted using AWS EC2 t2.micro instances and were connected to each other via internal VPCs when in the same region, using external connections between regions. The store, called Honu, is implemented in Go 1.9 using gRPC and protocol buffers for RPC requests; all code is open source and available on GitHub.

The workload on the system was generated by 15 clients, one in each region

and colocated with one of the replicas. Clients continuously created Put requests for random keys with a unique prefix per-region such that consistency conflicts only occur within a single region. The average throughput generated per-client was 5620.4 puts/second. The mean synchronization latency between each region ranged from 35ms to 630ms as shown in Figures 6.1 and 6.2. To ensure at least one synchronization per anti-entropy session, we set the anti-entropy interval to 1 second to train the system, then reduced the interval to 125ms while measuring visibility latency. To account for lag between commands sent to replicas in different regions, each experiment was run for 11 minutes, the bandit learning period was 4 minutes then visibility latency was observed for 6 minutes, buffered by 30 seconds before and after the workload to allow replicas to initialize and gracefully shutdown.

Our first experiments compared uniform random peer selection with epsilon greedy bandits using $\epsilon \in \{0.1, 0.2, 0.5\}$ as well as an annealing epsilon greedy bandit. The total system rewards as a rolling mean over a time window of 20 synchronizations are shown in Figure 6.3. The rewards ramp up from zero as the clients come online and start creating work to be synchronized. All of the bandit algorithms eventually improve over the baseline of uniform selection, not only generating more total reward across the system, but also introducing less variability in rewards over time. None of the bandit curves immediately produces high rewards as they explore the reward space; lower $\epsilon$ values may cause exploitation of incorrect arms, while higher $\epsilon$ values take longer to find optimal topologies. However, in the static workload case, the more aggressive bandit strategies converge more quickly to the optimal reward.

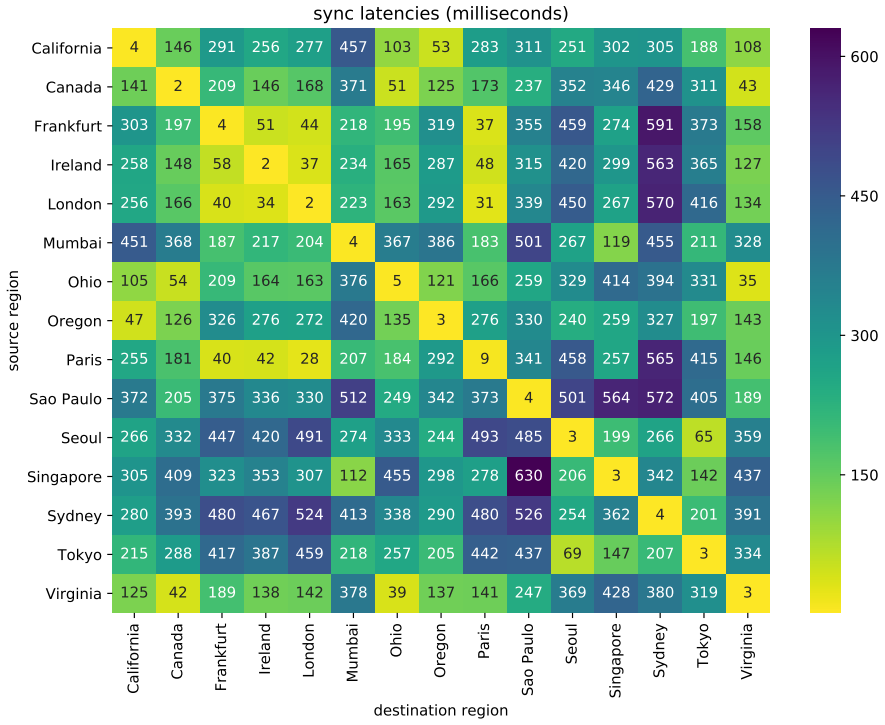


Figure 6.1: Inter-Region Synchronization Latencies (Push+Pull)

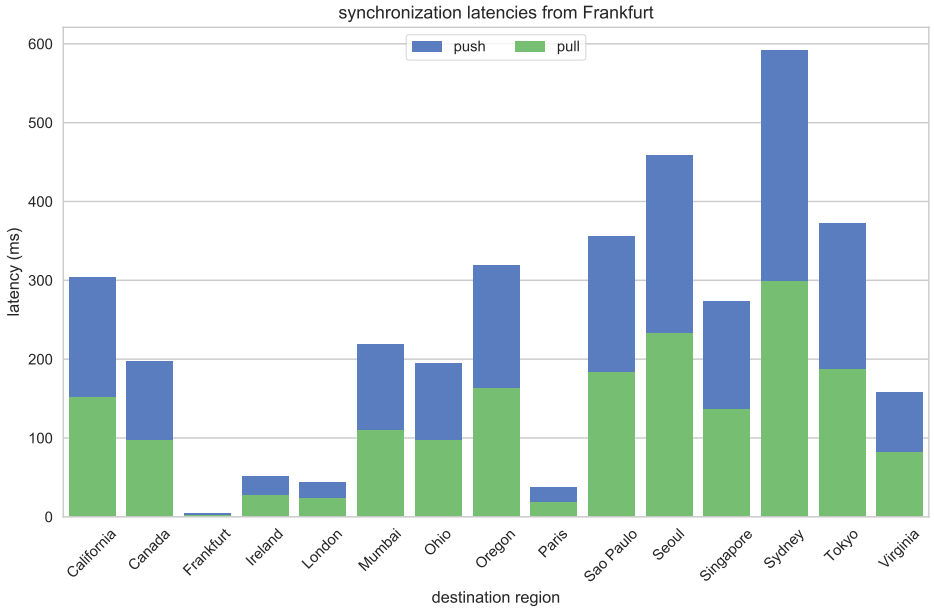


Figure 6.2: View of anti-entropy synchronization latency from Europe and corresponding network distances.

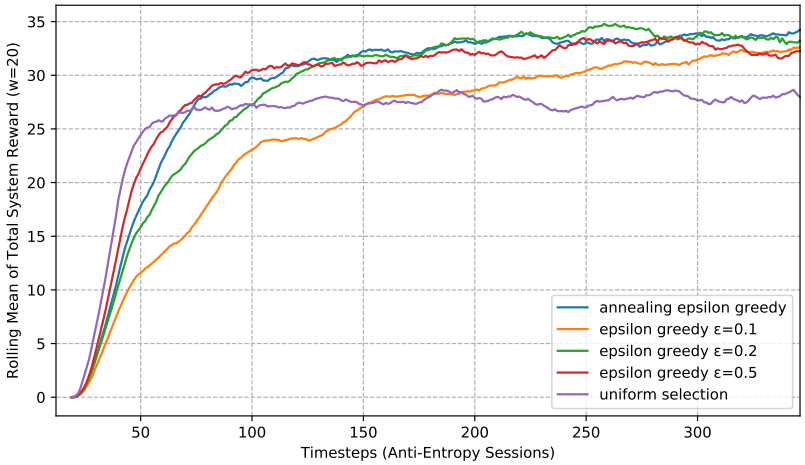


Figure 6.3: Total system rewards over time.

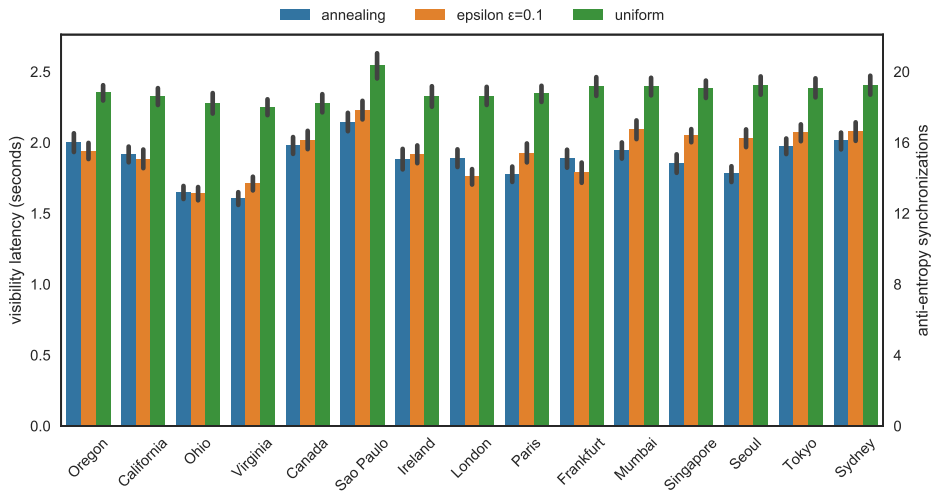


Figure 6.4: Decreasing visibility latency from bandit approaches.

Visibility latencies were computed by reducing the workload rate to once every 4 seconds to ensure the write becomes fully visible across the entire network. During the visibility measurement period, replicas locally logged the timestamp the write was pushed or pulled; visibility latency is computed as the difference between the minimum and maximum timestamp. The average visibility latency per region is shown in Figure 6.4 measured by the left y-axis. Because the anti-entropy delay is a fixed interval, the estimated number of required anti-entropy sessions associated with the visibility delay is shown on the right y-axis of the same figure. Employing bandit strategies reduces the visibility latency from 2360ms on average in the uniform case to 1870ms, reducing the number of required anti-entropy intervals by approximately 4.

To show the emergent behavior of bandits, we have visualized the resulting topologies as network diagrams in Figure 6.5 (uniform selection), Figure 6.7 (an-

nealing epsilon) and Figure 6.6 (epsilon greedy $\epsilon = 0.2$). Each network diagram shows each replica as a vertex, colored by region e.g. purple is California, teal is Sao Paulo, Brazil, etc. Each vertex is also labeled with the 2-character UN country or US state abbreviation as well as the replica's precedence id. The size of the vertex represents the number of `Put` requests that replica received over the course of the experiment; larger vertices represent replicas that were colocated with workload generators. Each edge between vertices represents the total number of successful synchronizations, the darker and thicker the edge is, the more synchronizations occurred between the two replicas. Edges are directed; the source of the edge is the replica that initiated anti-entropy with the target of the edge.

Comparing the resulting networks, it is easy to see that more defined topologies result from the bandit-based approaches. The uniform selection network is simply a hairball of connections with a limited number of synchronizations. By contrast, clear optimal connections have emerged with the bandit strategies; dark lines represent extremely successful synchronization connections between replicas, while light lines represent synchronization pairs that are selected less frequently. Based on our observations, we posit that fewer edges in the graph represents a more stable network; the fewer synchronization pairs that are selected, the less noise that occurs from selecting a peer that is in a similar state.

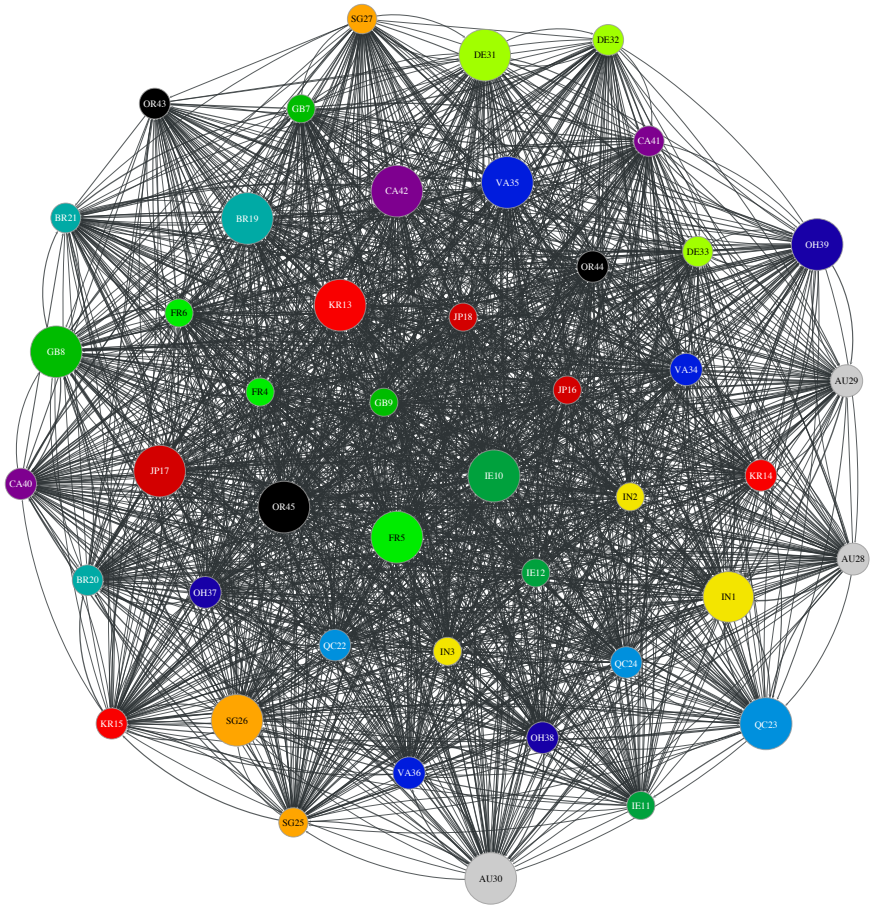


Figure 6.5: Synchronization network using uniform random selection of synchronization peers.

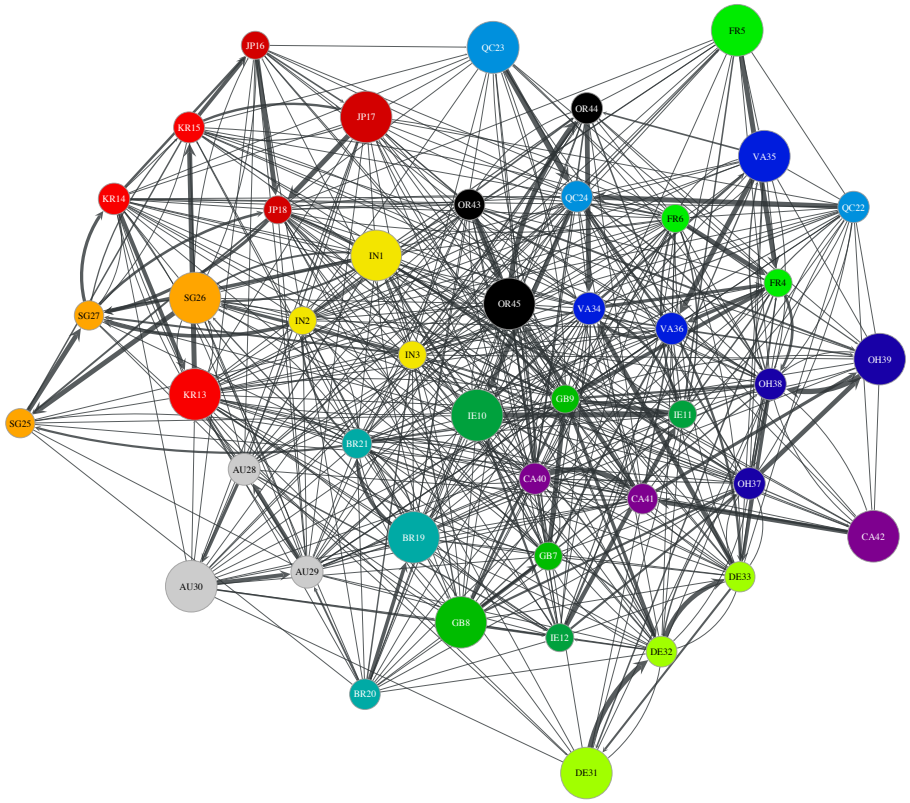


Figure 6.6: Synchronization network using bandit based selection of synchronization peers with $\epsilon = 0.2$.

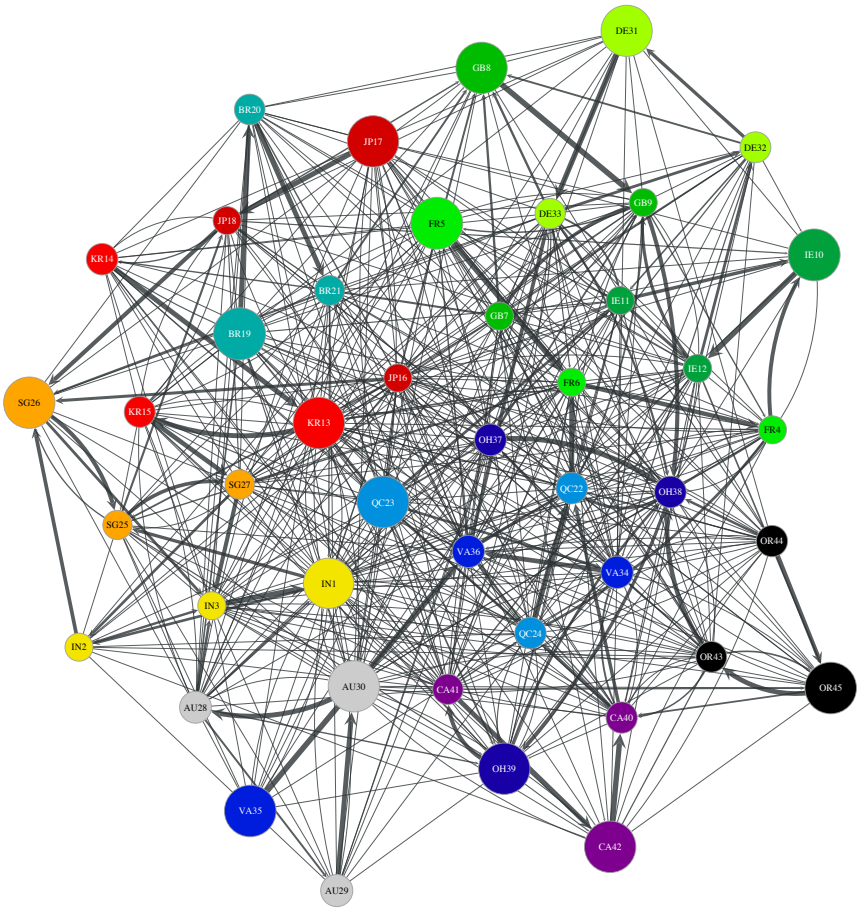


Figure 6.7: Synchronization network using annealing epsilon bandit based selection of synchronization peers.

## 6.2 Bandits Discussion

To achieve stronger eventual consistency, the visibility latency of a system replicated with anti-entropy must be reduced. We believe that this can be achieved with two primary goals: increasing the number of successful synchronizations and maximizing the number of local and regional synchronizations such that the average latency of anti-entropy sessions is as low as possible. These goals must also be tempered against other requirements, such as fault and partition tolerance, a deterministic anti-entropy solution that ensures the system will become consistent eventually, and load balancing the synchronization workload evenly across all replicas.

Bandit based approaches to peer selection clearly reduce noise inherent in uniform random selection as shown in Figure 6.3. The bandit strategies achieve better rewards over time because peers are selected that are more likely to have an update to synchronize. Moreover, based on the network diagrams shown in Figures 6.5-6.6, this is not the result of one or two replicas becoming primary syncs: most replicas have only one or two dark in-edges meaning that most replicas are only the most valuable peers for one or two other replicas.

Unfortunately, the rewards using a bandit approach, while clearly better than the uniform case, are not significantly better – this is an interesting demonstration of the possibility of adaptive systems to improve consistency but further investigation is required. The primary place we see for adjustment is future work to explore the reward function in detail. For example, the inclusion of penalties (negative rewards)

might make the system faster to adjust to a high quality topology. Comparing reward functions against variable workloads may also reveal a continuum that can be tuned to the specific needs of the system.

As for localization, there does appear to be a natural inclination for replicas that are geographically proximate to be a more likely selection. In Figure 6.6, replicas in Canada (light blue), Virginia (dark blue), Sydney (grey), California (purple), and Frankfurt (light green) all prioritize local connections. Regionally, this same figure shows strong links such as those between Ohio and California (`CA42` $\rightarrow$ `OH38`) or Japan and Singapore (`JP17` $\rightarrow$ `SG25`). Replicas such as `BR19` and `IN3` appear to be hubs that specialize in cross-region collaboration. Unfortunately there does also seem to be an isolating effect, for example Sydney (grey) appears to have no significant out of region synchronization partners. Isolated regions could probably be eliminated by scaling rewards with the number of transmitted updates, or by using larger epsilons. Multi-stage bandits might be used to create a tiered reward system to specifically adjust the selection of local, regional, and global peers. Other strategies such as upper confidence bounds, softmax, or Bayesian selection may also create more robust localization.

Finally, and perhaps most significantly, the experiments conducted in this paper were on a static workload; future work must explore dynamic workloads with changing access patterns to more closely simulate real world scenarios. While bandit algorithms are considered online algorithms that do respond to changing conditions, the epsilon greedy strategy can be slow to change since it prefers to exploit high-value arms. Contextual bandits use side information in addition to rewards to make

selection decisions, and there is current research in exploring contextual bandits in dynamic worlds that may be applicable [188]. Other strategies such as periodic reseting of the values may incur a small cost to explore the best anti-entropy topology, but could respond to changing access patterns or conditions in a meaningful way.

Future efforts will consider different reward functions, different selection strategies, dynamic environments, and how the priorities of system designers can be embedded into rewards. Reward functions that capture more information about the expected workload of the system such as object size, number of conflicts, or localizing objects may allow specific tuning of the adaptive approach. We will also specifically explore in detail the effect of dynamic workloads on the system and how the reinforcement learning can adapt in real time to changing conditions. We plan to investigate periodic resets, anomaly detection, and auction mechanisms to produce efficient topologies that are not brittle as access patterns change. We also plan to evaluate other reinforcement learning strategies such as neural or Bayesian networks to determine if they handle dynamic environments more effectively.

## 6.3 Access Temperature Approaches

add this section

- Expected model of access patterns (daylight).

## 6.4 Other Types of Adaptation

add this section and conclusion

Monitor and Optimize.

Replica placement, object placement

## 6.5 Conclusion

In this chapter we have presented a demonstration of adaptive consistency in the geo-replicated eventually consistent systems by employing a novel approach to peer selection during anti-entropy – replacing uniform random selection with multi-armed bandits. Multi-armed bandits consider the historical reward obtained from synchronization with a peer, defined by the number of objects synchronized and the latency of RPCs, when making a selection. Bandits balance the exploitation of a known high-value synchronization peer with the exploration of possibly better peers or the impact of failures or partitions. The end result is a replication network that is less perturbed by noise due to randomness and capable of more efficiently propagating updates.

In an eventually consistent system, efficient propagation of updates is directly tied to higher consistency. By reducing visibility latency, the likelihood of a stale read decreases, which is the primary source of inconsistency in a highly available system. We have demonstrated that bandit approaches do in fact lower visibility latency in a large network.

We believe that the results presented show a promising start to a renewed investigation of highly available distributed storage systems in novel network environments, particularly those that span the globe. Specifically, this work is part

of a larger exploration of adaptive, globally distributed data systems that federate consistency levels to provide stronger guarantees [189]. Federated consistency combines adaptive eventually consistent systems such as the one presented in this paper with scaling geo-replicated consensus such as Hierarchical Consensus [103] in order to create robust data systems that are automatically tuned to provide the best availability and consistency. Distributed systems that adapt to and learn from their environments and access patterns, such as the emerging synchronization topologies we observed in this paper, may form the foundation for the extremely large, extremely efficient networks of the future.

# Chapter 7: Related Work

Spanner [41] provides global consistency by sharding each tablet across multiple Paxos groups then externalizes their consistency using TrueTime, delaying the commit until a window of uncertainty has passed.

CalvinFS [43,44] batches transaction operations across the wide area, but still requires paxos to be deployed across the wide area.

Systems that implement many small quorums of coordination [41,42,57] avoid the centralization bottleneck and reliability concerns of master-service systems [65, 190] but create silos of independent operation that are not coordinated with respect to each other.

We have theorized that cloud services present the opportunity for deploying data services on a trusted infrastructure. If it is not trusted, however, we can use an encryption model similar to SPORC to combine multiple cloud providers into a single data system [191].

## 7.1 Hierarchical Consensus

Our principle contribution is Hierarchical Consensus, a general technique to compose consensus groups, maintain consistency invariants over large systems, and

adapt to changing conditions and application loads. HC is related to the large body of work improving throughput in distributed consensus over the Paxos protocol [60, 87, 91, 192], and on Raft [92, 146]. These approaches focus on fast vs. slow path consensus, eliding phases with dependency resolution, and load balancing.

Our work is also orthogonal in that subquorums and the root quorums can be implemented with different underlying protocols, though the two levels must be integrated quite tightly. Further, HC abstracts reconfiguration away from subquorum consensus, allowing multiple subquorums to move into new configurations and reducing the need for joint consensus [92] and other heavyweight procedures. Finally, its hierarchical nature allows the system to multiplex multiple consensus instances on disjoint partitions of the object space while still maintaining global consistency guarantees.

The global consistency guarantees of HC are in direct contrast to other systems that scale by exploiting multiple consensus instances [36,41,42] on a per-object basis. These systems retain the advantage of small quorum sizes but cannot provide system-wide consistency invariants. Another set of systems uses quorum-based decision-making but relaxes consistency guarantees [35, 37, 137]; others provide no way to pivot the entire system to a new configuration [57]. Chain replication [193] and Vertical Paxos [108] are among approaches that control Paxos instances through other consensus decisions. However, HC differs in the deep integration of the two different levels. Whereas these approaches are top down, HC consensus decisions at the root level replace system configuration at the subquorum level, and vice versa.

Possibly the closest system to HC is Scatter [57], which uses an overlay to orga-

nize consistent groups into a ring. Neighbors can join, split, and talk amongst themselves. The bottom-up approach potentially allows scaling to many subquorums, but the lack of central control makes it hard to implement global re-maps beyond the reach of local neighbors. HC ties the root quorum and subquorums tightly together, allowing root quorum decisions to completely reconfigure the running system on the fly either on demand or by detecting changes in network conditions.

We claim very strong consistency across a large distributed system, similar to Spanner [41]. Spanner provides linearizable transactions through use of special hardware and environments, which are used to tightly synchronize clocks in the distributed setting. Spanner therefore relies on a very specific, curated environment. HC targets a wider range of systems that require cost effective scaling in the data center to rich dynamic environments with heterogeneity on all levels.

Finally, shared logs have proven useful in a number of settings from fault tolerance to correctness guarantees. However, keeping such logs consistent in even a single consensus instance has proven difficult [58, 65, 73]. More recent systems are leveraging hardware support to provide fast access to shared logs [43, 44, 111, 113, 114, 194]. To our knowledge, HC is the first work to propose synchronizing shared logs across multiple discrete consensus instances in the wide area.

## 7.2 Federated Consistency

One of the earliest attempts to hybridize weak and strong consistency was a model for parallel programming on shared memory systems by Agrawal et al [133].

This model allowed programmers to relax strong consistency in certain contexts with causal memory or pipelined random access in order to improve parallel performance of applications. Per-operation consistency was extended to distributed storage by the RedBlue consistency model of Li et al [95]. Here, replication operations are broken down into small, commutative sub-operations that are classified as red (must be executed in the same order on all replicas) or blue (execution order can vary from site to site), so long as the dependencies of each sub-operation are maintained. The consistency model is therefore global, specified by the red/blue ordering and can be adapted by redefining the ratio of red to blue operations, e.g. all blue operations is an eventually consistent system and all red is sequential.

The next level above per-operation consistency hybridization is called *consistency rationing* wherein individual objects or groups of objects have different consistency levels applied to them to create a global quality of service guarantee. Kraska et al. [130] initially proposed consistency rationing be on a per-transaction basis by classifying objects in three tiers: eventual, adaptable, and linearizable. Objects in the first and last groups were automatically assigned transaction semantics that maintained that level of consistency; however objects assigned the adaptable categorization had their consistency policies switched at runtime based on a cost function that either minimized time or write costs depending on user preference. This allowed consistency in the adaptable tier to be flexible and responsive to usage.

Chihoub et al. extended the idea of consistency rationing and proposed limiting the number of stale reads or the automatic minimization of some consistency

cost metric by using reporting and consistency levels already established in existing databases [128,129]. Here multiple consistency levels are being utilized, but only one consistency model is employed at any given time for all objects, relaxing or strengthening depending on observed costs. By utilizing all possible consistency semantics in the database, this model allows a greater spectrum of consistency guarantees that adapt at runtime.

Al-Ekram and Holt [195] propose a middleware based scheme to allow multiple consistency models in a single distributed storage system. They identify a similar range of consistency models, but use a middleware layer to forward client requests to an available replica that maintains consistency at the lowest required criteria by the client. However, although their work can be extended to deploying several consistency models in one system, they still expect a homogeneous consistency model that can be swapped out on demand as client requirements change. Additionally their view of the ordering of updates of a system is from one versioned state to another and they apply their consistency reasoning to the divergence of a local replica's state version and the global version. Similar to SUNDR, proposed by Li et al. [196], an inconsistency is a fork in the global ordering of reads and writes (a "history fork"). Our consistency model instead considers object forks, a more granular level that allows concurrent access to different objects without conflict while still ensuring that no history forks can happen.

Hybridization and adaptation build upon previous work that strictly categorizes different consistency schemes. An alternative approach is to view consistency along a continuous scale with several axes that can be tuned precisely. Yu and

Vahdat [197] propose the *conit*, a consistency unit described as a three dimensional vector that describes tolerable deviations from linearizability along staleness, order error, and numeric ordering. Similarly, Afek et al. [198] present quasi-linearizable histories which specify a bound on the relative movement of ordered items in a log which make it legally sequential.

A strong central core to provide support to the entire system has been suggested both in Oceanstore [1] and primary copy schemes [108, 190]. We take this idea further in our experiments by Federating a strong central core composed of Replicas that perform consensus via the Raft consensus protocol and combine it with highly available eventually consistent systems. In this way Federated consistency gains the flexibility and availability of the Eventual replicas (leader re-election and no requirement for remote writes mean that the replica can continue even if it is completely partitioned from the rest of the network) while still getting guarantees from Raft, which minimizes "fork flipping" – the behavior of writing to one branch then another, truly pernicious inconsistent behavior that cannot be prevented in an eventually consistent system.

System designers can take advantage of heterogeneous replicas by implementing stronger consistency on more reliable machines that are able to handle more messages. Mobile replicas that are prone to network loss, out of order or missed messages, or other variable behavior can adapt their policy depending on the environment they're in. Our model also allows for *adaptive* behavior, in that the replicas can monitor the environment for change and as the mean latency decreases, adapt their $T$ parameter accordingly. This is a no cost operation for Eventual repli-

cas (who can also optimize pairwise gossip by implementing non-discrete random selection using Bandits or other optimization techniques), and only requires joint consensus on the part of the consensus group.

# Chapter 8: Conclusion

# Appendix A: Formal Specification

Will add formal specification here.

# Bibliography


[1] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, and others. Oceanstore: An architecture for global-scale persistent storage. In *ACM Sigplan Notices*, volume 35, pages 190–201.

[2] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Elsevier.

[3] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium On*, pages 124–131. IEEE.

[4] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. 2014(239):2.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. A view of cloud computing. 53(4):50–58.

[6] Cisco Visual Networking Index. The zettabyte era–trends and analysis.

[7] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *Information Processing in Sensor Networks, 2009. IPSN 2009. International Conference On*, pages 349–360.

[8] Michael P. Andersen, Sam Kumar, Connor Brooks, Alexandra von Meier, and David E. Culler. DISTIL: Design and implementation of a scalable synchrophasor data processing system. In *Smart Grid Communications (SmartGridComm), 2015 IEEE International Conference On*, pages 271–277. IEEE.

[9] Lars Wischoff, André Ebner, Hermann Rohling, Matthias Lott, and Rüdiger Halfmann. SOTIS-a self-organizing traffic information system. In *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, volume 4, pages 2442–2446.

[10] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of Things: Vision, Applications and Research Challenges. 10(7):1497–1516.

[11] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM.

[12] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: The Consumers' Perspective. In *CIDR*, volume 11, pages 134–143.

[13] Sérgio Esteves, Joao Silva, and Luís Veiga. Quality-of-service for consistency of data geo-replication in cloud computing. In *European Conference on Parallel Processing*, pages 285–297.

[14] Ravi Jhawar, Vincenzo Piuri, and Marco Santambrogio. Fault tolerance management in cloud computing: A system-level perspective. 7(2):288–297.

[15] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. 32(2):374–382.

[17] Michael J. Casey and Paul Vigna. In blockchain we trust. 15:2018.

[18] Michael Stonebraker and Joey Hellerstein. What goes around comes around. 4.

[19] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: Friends or foes? 53(1):64–71.

[20] C. Mohan. History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 11–16. ACM.

[21] Erik Kain. The 'Pokémon GO' Launch Has Been A Complete Disaster [Updated].

[22] Cloud Datastore.

[23] Eric A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 167–167. ACM.

[24] Luke Stone. Bringing Pokémon GO to life on Google Cloud.

[25] Makiko Yamazaki. Developer of Nintendo's Pokemon GO aiming for rollout to 200...

[26] Richard George. Nintendo's President Discusses Region Locking.

[27] Dropbox — Company Info.

[28] Slack. Slack About Us.

[29] WeWork. Global Access.

[30] Tile About Tile.

[31] Stella Garber. Lessons Learned From Launching Internationally.

[32] Desdemona Bandini. RunKeeper Scales to Meet Demand from 24 Million Global Users with New Relic.

[33] Adam Grossman and Jay LaPorte. Dark Sky Weather App for iOS and Android.

[34] Matthew Hughes. Signal and Telegram are growing rapidly in countries with corruption problems.

[35] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. 1(2):1277–1288.

[36] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. 26(2):4.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM.

[38] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. 44(2):35–40.

[39] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. pages 22–28.

[40] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234.

[41] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and others. Spanner: Google's globally distributed database. 31(3):8.

[42] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM.

[43] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM.

[44] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14.

[45] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. 33(2):51–59.

[46] Eric Brewer. Pushing the cap: Strategies for consistency and availability. 45(2):23–29.

[47] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations (Extended Version). 7(3):181–192.

[48] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. 45(2):37–42.

[49] Eric Brewer. CAP twelve years later: How the" rules" have changed. 45(2):23–29.

[50] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM.

[51] stevestein. Scaling out with Azure SQL Database.

[52] Alain Jobart, Sugu Sougoumarane, Michael Berlin, and Anthony Yeh. Vitess.

[53] Spencer Kimball, Peter Mattis, and Ben Darnell. CockroachDB.

[54] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards IP Geolocation Using Delay and Topology Measurements. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 71–84. ACM.

[55] A. Khiyaita, H. El Bakkali, M. Zbakh, and Dafir El Kettani. Load balancing cloud computing: State of art. In *Network Security and Systems (JNS2), 2012 National Days Of*, pages 106–109. IEEE.

[56] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 31(4):149–160.

[57] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM.

[58] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association.

[59] Blake Mizerany, Qin Yicheng, and Li Xiang. Etcd: Package raft.

[60] Leslie Lamport. The part-time parliament. 16(2):133–169.

[61] Butler W. Lampson. How to build a highly available system using consensus. In *Distributed Algorithms*, pages 1–17. Springer.

[62] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. LogBase: A scalable log-structured database system in the cloud. 5(10):1004–1015.

[63] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. 10(1):26–52.

[64] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. Log-structured memory for DRAM-based storage. In *FAST*, pages 1–16.

[65] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM.

[66] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and others. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 383–398.

[67] Dhruba Borthakur. HDFS architecture guide. 53:1–13.

[68] Andrew Fikes. Storage architecture and challenges.

[69] Michael P. Andersen and David E. Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *FAST*, pages 39–52.

[70] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 307–320. USENIX Association.

[71] Garth Alan Gibson. Redundant disk arrays: Reliable, parallel secondary storage.

[72] Sameer Wadkar and Madhu Siddalingaiah. Apache ambari. In *Pro Apache Hadoop*, pages 399–401. Springer.

[73] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9.

[74] David Bermbach and Jörn Kuhlenkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer.

[75] Werner Vogels. Eventually consistent. 52(1):40–44.

[76] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. 22(4):299–319.

[77] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Prentice-Hall.

[78] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference On*, pages 140–149. IEEE.

[79] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*.

[80] Mustaque Ahamad, Gil Neiger, Prince Kohli, James Burns, Phil Hutto, and T. E. Anderson. Causal memory: Definitions, implementation and programming. 1:6–16.

[81] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. 12(2):91–122.

[82] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. 100(9):690–691.

[83] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. 12(3):463–492.

[84] Leslie Lamport. Paxos made simple. 32(4):18–25.

[85] David Mazieres. Paxos made practical.

[86] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM.

[87] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM.

[88] Leslie Lamport. Fast paxos. 19(2):79–103.

[89] Martin Biely, Zoran Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium On*, pages 111–120. IEEE.

[90] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 316–317. ACM.

[91] Leslie Lamport. Generalized consensus and Paxos.

[92] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319.

[93] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. 5(8):776–787.

[94] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. 23(2):279–302.

[95] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278.

[96] Diego Didona, Kristina Spirovska, and Willy Zwaenepoel. The design of Wren, a Fast and Scalable Transactional Causally Consistent Geo-Replicated Key-Value Store.

[97] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM.

[98] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), Lombard, IL*.

[99] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M Hellerstein. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 23. ACM.

[100] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, pages 177–190.

[101] Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324. ACM.

[102] Amazon Simple Storage Service (Amazon S3).

[103] Benjamin Bengfort and Pete Keleher. Brief Announcement: Hierarchical Consensus. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing*, pages 355–357. ACM.

[104] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, volume 8, pages 369–384.

[105] Pierre Sutra and Marc Shapiro. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium On*, pages 255–264. IEEE.

[106] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference On*, pages 156–167. IEEE.

[107] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-stop Processors. 2(2):145–154.

[108] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pages 312–313. ACM.

[109] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI*, volume 4, pages 8–8.

[110] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. 3(4):1.

[111] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14. ACM.

[112] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*, pages 1–14.

[113] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, and others. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *NSDI*, pages 35–49.

[114] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM.

[115] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 3. ACM.

[116] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. 4(3):382–401.

[117] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. 29.

[118] David K. Gifford. Information Storage in a Decentralized Computer System.

[119] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7):558–565.

[120] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaram, Daniel J. Abadi, James Aspnes, Sen Siddhartha, and Mahesh Balakrishnan. The FuzzyLog: A Partially Ordered Shared Log.

[121] Minkyong Kim, Landon P. Cox, and Brian D. Noble. Safety, Visibility, and Performance in a Wide-Area File System. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*.

[122] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM.

[123] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. 45(5):37–42.

[124] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. 44(5):27–32.

[125] Tom H. Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. Fog computing: Focusing on mobile users at the edge.

[126] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, pages 13–16. ACM.

[127] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 263–270. ACM.

[128] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301. IEEE.

[129] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Consistency in the cloud: When money does matter! In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium On*, pages 352–359. IEEE.

[130] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. 2(1):253–264.

[131] Evaggelia Pitoura and Bharat Bhargava. Data consistency in intermittently connected distributed systems. 11(6):896–915.

[132] U. Cetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin. Deno: A Decentralized, Peer-to-Peer Object Replication System for Mobile and Weakly-Connected Environments. 52(7).

[133] Divyakant Agrawal, Manhoi Choy, Hong Va Leong, and Ambuj K. Singh. Mixed consistency: A model for parallel programming. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 101–110. ACM.

[134] Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM.

[135] Michel Raynal. Sequential consistency as lazy linearizability. In *EurAsia-ICT 2002: Information and Communication Technology*, pages 866–873. Springer.

[136] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency.

[137] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM.

[138] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 International Conference on Management of Data*, pages 761–772. ACM.

[139] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. (3):240–247.

[140] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps-decentralized version vectors. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference On*, pages 544–551. IEEE.

[141] Bernhard Haeupler. Simple, fast and deterministic gossip and rumor spreading. 62(6):47.

[142] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium On*, pages 482–491. IEEE.

[143] Joao Leitao, José Pereira, and Luis Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference On*, pages 419–429. IEEE.

[144] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer.

[145] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. 25(14):1754–1760.

[146] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: Do we have consensus? 49(1):12–21.

[147] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the Internet. 39(5):22–31.

[148] Gul A. Agha. Actors: A model of concurrent computation in distributed systems.

[149] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. 410(2):202–220.

[150] P. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability.

[151] Carl Hewitt. Viewing control structures as patterns of passing messages. 8(3):323–364.

[152] Michael Stonebraker. The case for shared nothing. 9(1):4–9.

[153] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. 2:199–216.

[154] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246.

[155] Tim Dierks. The transport layer security (TLS) protocol version 1.2.

[156] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer.

[157] Levent Ertaul, Sarika Singhal, and Gökay Saldamli. Security Challenges in Cloud Computing. In *Security and Management*, pages 36–42.

[158] Robert Griesemer, Rob Pike, and Ken Thompson. The Go programming language.

[159] Charles Antony Richard Hoare. Communicating sequential processes. 21(8):666–677.

[160] Google. Google RPC.

[161] Protocol Buffers: Google's Data Interchange Format.

[162] Benjamin Bengfort. Alia.

[163] Benjamin Bengfort. Raft.

[164] Benjamin Bengfort. Honu.

[165] Sanjay Ghemawat and Jeff Dean. LevelDB, A fast and lightweight key/value database library by Google.

[166] BadgerDB: Fast key-value DB in Go.

[167] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM.

[168] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. 9(3):9.

[169] Jon Gjengset <jon@thesquareplanet.com>. Students' Guide to Raft :: Jon Gjengset.

[170] Caio Oliveira, Lau Cheuk Lung, Hylson Netto, and Luciana Rech. Evaluating raft in docker on kubernetes. In *International Conference on Systems Science*, pages 123–130. Springer.

[171] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. 6(1):51–81.

[172] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*.

[173] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187.

[174] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 127–140.

[175] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium On*, pages 1–10. IEEE.

[176] Robert B Ross, Rajeev Thakur, and others. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430.

[177] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. GlobalFS: A Strongly Consistent Multi-Site File System.

[178] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. 31(2):249–260.

[179] Michael O. Rabin. Fingerprinting by Random Polynomials.

[180] Pat Helland. Immutability changes everything. 13(9):40.

[181] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

[182] Benjamin Bengfort, Philip Y. Kim, Kevin Harrison, and James A. Reggia. Evolutionary design of self-organizing particle systems for collective problem solving. In *Swarm Intelligence (SIS), 2014 IEEE Symposium On*, pages 1–8. IEEE.

[183] Benjamin Bengfort, Konstantinos Xirogiannopoulos, and Pete Keleher. Anti-Entropy Bandits for Geo-Replicated Consistency. In *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press.

[184] Salvatore Scellato, Cecilia Mascolo, Mirco Musolesi, and Jon Crowcroft. Track globally, deliver locally: Improving content delivery networks by tracking geographic social cascades. In *Proceedings of the 20th International Conference on World Wide Web*, pages 457–466. ACM.

[185] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium On*, pages 565–574. IEEE.

[186] Yamir Moreno, Maziar Nekovee, and Amalio F. Pacheco. Dynamics of rumor spreading in complex networks. 69(6):066130.

[187] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information. In *Advances in Neural Information Processing Systems*, pages 817–824.

[188] Haipeng Luo, Alekh Agarwal, and John Langford. Efficient Contextual Bandits in Non-stationary Worlds.

[189] Benjamin Bengfort and Pete Keleher. Federating Consistency for Partition-Prone Networks. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE.

[190] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *ACM SIGMOD Record*, volume 25, pages 173–182. ACM.

[191] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration using Untrusted Cloud Resources. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 337–350. USENIX Association.

[192] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited.

[193] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, volume 4, pages 91–104.

[194] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, volume 11, pages 9–12.

[195] Raihan Al-Ekram and Ric Holt. Multi-consistency data replication. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference On*, pages 568–577. IEEE.

[196] Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis E. Shasha. Secure Untrusted Data Repository (SUNDR). In *OSDI*, volume 4, pages 9–9.

[197] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. 20(3):239–282.

[198] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer.